PC al límite

PROGRAMACION AVANZADA

Boris Bertelsons
Mathias Rasch

- Aplicaciones prácticas avanzadas en Ensamblador y Turbo Pascal
- Programación de gráficos en 3D
- Cuerpos transparentes
- Controladores PIT, DMA y RTC

CONTIENE CD-ROM

- Código de los programas
- Los trucos y secretos de los profesionales
- Herramientas shareware

Computec
BOIXAREU EDITORES
marcombo
DATA BECKER EDITION

PROGRAMACION AVANZADA

PC al límite

PROGRAMACION AVANZADA

PC al límite

Boris Bertelsons
Mathias Rasch

Computec
marcombo

Programación avanzada, PC al límite
B. Bertelsons - M. Rasch
Traducción autorizada de la edición original en lengua alemana
por DATA BECKER, GmbH-Düsseldorf

Derechos reservados en lengua española conforme a la ley
© 1995 Marcombo, S.A., Barcelona, España

© 1998 ALFAOMEGA GRUPO EDITOR, S.A. de C.V.
Apartado Postal 73-267, 03311 México, D.F.

Miembro de la Cámara Nacional de la Industria Editorial
Registro No. 2317

ISBN 970-15-0085-7

Derechos reservados.
Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Edición autorizada para venta en México, Colombia, Ecuador, Perú, Bolivia, Chile, Centroamérica, Estados Unidos y el Caribe.

Impreso en Colombia–Printed in Colombia

NOTA IMPORTANTE

La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro y en el CD-ROM adjunto, ni por la utilización indebida que pudiera dársele.

Prólogo

Aquellos que hace dos años pensaban en costosos mapeados de texturas, plasmas de hardware, calidad de sonido de 16 bit o sombreados Gouraud, y que se imaginaban todo esto unido en presentaciones gráficas, seguramente se planteaban la pregunta si sería posible algún día. La respuesta solía ser: "seguro que no", y se seguía soñando con algo imposible.

Pero los tiempos han cambiado (afortunadamente). Para todo aquel que se crea algo, hoy en día estas técnicas ya son obligatorias y no se contentará con menos. Las exigencias a los programadores han crecido por lo tanto considerablemente.

Cuando antes ya se estaba contento con una buena programación de efectos y no se valoraban otros aspectos, hoy en día nos enfrentamos con un público cada vez más numeroso que pide más y más y lo exige de forma clara. Por esta razón ya no basta apostar solamente sobre pura magia efectista, hoy en día se precisa un equipo completo de programadores, músicos, grafistas y hasta diseñadores, apurando los límites máximos de sus conocimientos. Una "demo" ya no es un proyecto de 2 días, sino que precisa su tiempo de la misma forma que la programación de juegos y aplicaciones.

Nadie se hubiera imaginado una evolución de este tipo sobre la base de un PC, más aún cuando se le consideraba una máquina de oficina y no un ordenador para todos los usos, que hoy en día ya está presente en toda Europa en uno de cada cuatro hogares y cuya marcha triunfal no se va a frenar en mucho tiempo. Porque gracias a su flexibilidad, el ordenador personal se ha convertido en la base de toda la evolución.

Hoy en día muchos usuarios de otros sistemas de ordenadores miran con recelo a su compañero del PC. La imagen de multitalento del PC no solo se debe a la técnica que ha avanzado constantemente, sino también a las ganas de experimentar de tantos usuarios entusiastas, entre los cuales se encuentran por ley natural muchas personas que no se contentan con las posibilidades actuales. Por esta razón no es difícil de explicar que los gigantes del software recurran hoy en día cada vez más para la realización de sus proyectos a Equipos de Demo. (Esto es un simple ejemplo.)

A los freaks no les interesa primordialmente la ganancia económica, sino que luchan por el prestigio y el reconocimiento que se suele conseguir con buenos proyectos. Pero hay que tener cuidado: aquel que desee entrar en este juego deberá estar al tanto de lo que habla. Porque la dedicación es la que ha convertido a los

freaks en lo que representan hoy en día para muchas personas. Sin su experiencia como pioneros y las muchas horas de trabajo la técnica que hoy se denomina sin la más mínima duda como estándar, estaría hoy en día bastante más atrasada.

¿ Por qué me debería dedicar a este tema ?

Esta pregunta tiene su justificación, aunque los autores no duden ni un instante sobre la respuesta. La programación de las tarjetas gráficas de los PC ya tiene bastante historia. Pero al introducirse la tarjeta VGA como "quasi-estándar", la situación cambió bastante. Por fin se tenía la posibilidad de utilizar el PC para algo más que para la introducción y reproducción de textos. Las capacidades de la tarjeta VGA permitieron rápidamente complejas operaciones gráficas. Estas fueron aprovechadas por nosotros para crear efectos ópticamente impresionantes. Muy ligado a todo ello están la música y el sonido. La utilización de tarjetas de sonido se puede considerar como un avance que se ha producido en tiempos recientes.

Todo esto comenzó con la llamada tarjeta AdLib, que aún hoy en día tiene algunas aplicaciones. Lógicamente la gente no se contentó durante demasiado tiempo con los sonidos generados con la FM, sino que se añadió la posibilidad del muestreo (sampling). Hoy en día las tarjetas de sonido ya se suministran con las llamadas wave-tables, cuyas cualidades son comparables a las de los estudios de grabación. La programación de las tarjetas de sonido, ya sea mediante la síntesis FM o de wave-tables comenzó siendo accesoria, pero es también ciertamente fascinante, ya que la utilización conjunta de la óptica y los efectos sonoros es lo que consigue la armonía en estos proyectos.

Aunque nos hayamos ido un poco del tema, vamos a hablar un poco del contenido de este libro. Esta obra pretende acercar al lector /lectora a toda esta temática que sin duda alguna sigue siendo bastante compleja.

Ha sido realizado y montado por personas que tienen bastantes conocimientos del llamado "ambiente", por lo que el título del libro queda bastante claro. De hecho nos vamos a dedicar en un ejemplo a la programación de vectores, que en su recorrido serán rellenados por las llamadas texturas. La utilización de la técnica del mapeado de texturas es a primera vista bastante complicada, pero vale la pena dedicarse un poco, ya que hoy en día es muy popular y permite generar efectos de una realidad jamás soñada.

La utilización adicional de música y efectos sonoros a la que hacíamos referencia anteriormente encuentra su punto clave en la programación de reproductores MOD. El formato MOD para los archivos de música es el estándar en la actualidad, y en cuanto se entiende este conocimiento es trasladable sin ningún tipo de problemas a otros formatos. Lógicamente todo esto no es más que una pequeña

muestra de lo que se va a encontrar en este libro. Así hablaremos también de la programación de Voxel, Vectores Glenz, Fire y los procedimientos de protección por palabras clave (passwords). Otra parte interesante es la programación de entrenadores de juegos.

¿ Qué me espera ?

Como ya hemos explicado largo y tendido el presente libro llamado "PC AL LÍMITE" no es un manual usual. Aquel que busque soluciones a sus aplicaciones o algoritmos válidos globalmente no encontrará lo que busca. Lo que intenta por el contrario este libro es acercar al lector la fascinación que aún hoy en día roba el sueño a muchos "gurús". Partimos de la base que se poseen conocimientos básicos de la programación en Pascal y Ensamblador, ya que para la comprensión de ciertos temas son imprescindibles. Esto no significa que los principiantes se retiren precipitadamente, ya que esta obra siempre servirá para ir aprendiendo.

Que se diviertan.

Agradecimientos:

AccesDenied, AD, Admire, Alpha, Anarchy, Angelika, Atan, AXL, Blank, Cascada, Daryl, Dose, Dust, Ela, Epical, Extreme, FrankB., Gollum, Grif, Iguana, Imp/Cda, Impact, Ines, Infiny, Killer Voop, KLF, Legend Design, Majic 12, Mental Design, Moppel, Osti & Co., Pogo, Roger, Sandra, Silicon Worlds, Soft Ones, SSL, Stepp-Vieh, Surprise! Prods., THE COEXISTENCE & Xography, Tran, Tritón, Vera, VRS, Wolfang.

Índice general

1. Acerca de bytes y phreaking - el lenguaje del mundillo 17

2. El verdadero lenguaje de los programadores del mundillo: ensamblador 19

2.1 Multiplicaciones y divisiones en Ensamblador 20
2.2 Cálculos con coma fija 21
Las cuatro operaciones de cálculo básicas 22
Un ejemplo de aplicación 26
2.3 Funciones matemáticas propias 28
Tablas - ¡Que vivan las tablas! 29
La aproximación 31
2.4 Optimizar comparaciones 34
OR en vez de CMP 34
Comparaciones con desbordamiento 35
Comparaciones de cadenas 35
2.5 Variables en ensamblador 35
Acceso a variables de Pascal 36
Acceso a arrays y records 36
Variables de segmento de código (codesegment) 37
Arrays circulares 38
Enmascaramiento rotatorio de bits 38
Enmascaramiento de un número determinado de bits 39
2.6 El secreto de los cracks - las interrupciones 39
Desviar vectores 40
Llamada al controlador original y finalización 40
Bloqueo de interrupciones 41
Retorno al DOS - Reentrance 42
La captura de CTRL-C y Reset 43
2.7 Consejos para la programación de bucles 47
Anidar 47
Accesos de 16/32 bits 47
2.8 Instrucciones prácticas del 386 48
Las instrucciones MOVSX y MOVZX 48
Diferentes instrucciones SET 48
Multiplicaciones y divisiones rápidas con las instrucciones SHRD y SHRL 49
Multiplicación ampliada con IMUL 49
Utilización de instrucciones 386 en programas Pascal 50

3. Esto arranca: ensamblador en la práctica 51

3.1 El puerto paralelo 51
Programación directa del puerto paralelo 51
Acceso a la impresora 53
3.2 Otras aplicaciones 53

4. El conocimiento de los gráficos desde el submundo ... 57

4.1 Conceptos técnicos que hay que conocer 57
4.2 La base - modo 13h del BIOS 60
Organización de la memoria 60
La administración interna del modo 13h 61
4.3 El formato de imagen GIF 62
El estándar del mundillo 62
Cargador GIF optimizado a 320 x 200 67
PCX - simple y rápido 79
4.4 VGA hasta el último bit 89
El controlador del tubo de rayos catódicos (CRTC) 91
Timing Sequenzer (TS) 102
Graphics Data Controller (GDC) 105
Attribute Controller (ATC) 109
Digital to Analog Converter (DAC) 113

5. Modo X - el secreto mejor guardado 115

5.1 El poder gráfico del modo X 115
5.2 Inicialización 116
5.3 Estructura 117
5.4 Mayores resoluciones en el Modo X 119
5.5 La utilización del modo X 121
Situar puntos 122
Cambio de página 122
5.6 Ampliación del cargador GIF al Modo X 130
5.7 Un simple scrolly de texto 132
Un scrolly en el modo X 132
Juego de caracteres para el texto en movimiento 133

6. División de pantalla y otros efectos 135

6.1 Lo básico 135
6.2 Así funciona: división de pantalla 137
6.3 Scrolling en las cuatro direcciones 142
6.4 Todo a la vez - división de pantalla en combinación con el scrolling ... 149
6.5 Cerrar una puerta - juntar una imagen 151
6.6 Scrolling suave en el modo texto 152

6.7 Limpiar el monitor de una forma diferente - una imagen que se diluye 156
6.8 Un poco más de color, por favor - barras Copper sin copiar 158
6.9 Un flan en la pantalla - el Wobbler 165
6.10 Animaciones a tiempo real de forma fácil - efectos de paleta 169
Fundido a negro 170
El efecto contrario - fundir desde negro 172
El camino general - fundido desde cualquier fuente a la paleta destino 173
Como en las películas - fundido de una imagen a otra 177
La alternativa rápida - animación mediante rotación de paletas 190
6.11 El monitor en llamas - fuego 195
6.12 El secreto de Comanche (Tm) - espaciado Voxel 200

7. Sprites - acción rasante en la pantalla 207

7.1 Los fundamentos 207
7.2 Leer y escribir sprites 209
7.3 Más allá de todas las fronteras - Clipping 210
7.4 La unidad sprite 211
7.5 Para movimientos realistas - escalado 222

8. La tercera dimensión - programación gráfica 3D 229

8.1 Matemáticas para fanáticos de los gráficos 229
8.2 Representar cuerpos 3-D en 2-D 233
8.3 Deformar objetos - transformaciones 235
8.4 Modelos de alambre 237
8.5 Vea a través - cuerpos transparentes 257
8.6 Ocultar líneas - Hidden Line 273
8.7 Proyectar sombras - sombreado con luces 277
8.8 Superficies impresionantes - texturas 283

9. Protección moderna contra copias - la clave de acceso 295

9.1 Así protege sus programas - claves de acceso 295
9.2 Programas de lenguajes avanzados a nivel de máquina 306
9.3 Una clave de acceso protegida 311

10. Proteja sus conocimientos - trucos anti-debugging ... 319

10.1 Inspeccione programas 319
Abajo con el camuflaje - programas de descompresión 320
Paso a paso - Turbo Debugger 323
¡Hexpléndido! - editores hexadecimales 333
10.2 Las interrupciones de depuración 335
Enmascarar interrupciones 336

La desviación de las interrupciones de debugging 336
Guardar datos en el vector 336
Sustituir interrupciones 338
10.3 Engañar al debugger 339
El truco de la memoria 339
Instrucciones de doble sentido 339
Detener el TD386 340
10.4 Automodificaciones 341
Empleo correcto de las sumas de control 341
Algoritmos de encriptación 341
El truco PIQ 342
El empleo de programas compresores 343
10.5 Entrénese - entrenadores y depuradores de juegos 344
¡Busque! - encontrar variables 344
Manejo de los valores delimitadores 346
Así se programan entrenadores 347
Entrenador para el juego RAIDER 349

11. Administración de la memoria 359

11.1 La memoria DOS convencional 359
11.2 EMS - un paso adelante para más memoria 361
Las funciones del EMS 362
El empleo del EMS 366
11.3 XMS - Y tuya sea la memoria 372
Los códigos de error XMS 373
La programación XMS 374
11.4 El modelo de memoria Flat - la solución de sus problemas de memoria 381
El trasfondo técnico del modelo Flat 382
Así se programa el modelo Flat 383

12. Sáquele partido a su ordenador - programación de los componentes auxiliares 391

12.1 El concepto de interrupciones 391
12.2 El Programmable Interval Timer (PIT) 395
El hardware del PIT 395
El empleo del PIT 398
12.3 El Programmable Interrupt Controller (PIC) 402
12.4 El controlador DMA 404
El enmascarado del canal DMA 405
El ajuste del modo de transferencia 406
El borrado del Flip-Flop DMA 408
Ajustar la dirección del bloque de datos 408
Ajustar el tamaño de una transferencia DMA 409

12.5 El reloj de tiempo real (RTC) ... 412
Las funciones de reloj ... 413
Los registros de estado ... 414
Los bytes de configuración ... 415
El RTC en la práctica ... 419

13. Descubra un mundo de sonidos - la tarjeta Sound-Blaster ... 425

13.1 Componentes de las tarjetas Sound-Blaster ... 425
Programar el procesador digital de señales - DSP ... 425
Reconocimiento de la tarjeta Sound-Blaster ... 439
Mezclar datos de sonido - chip mezclador ... 441
13.2 Reproducción de archivos VOC con programación directa ... 452

14. Gravis Ultrasound - la tarjeta de sonido de los fanáticos ... 457

14.1 Esta es la estructura de la Gravis Ultrasound ... 457
14.2 Modo de funcionamiento de la GUS ... 458
14.3 Registros de la GUS ... 459
14.4 Programación de la GUS ... 474
Así se inicializa la GUS ... 474
Cargar datos de sonido ... 480
Reproducir Muestreos ... 482
Modificar voces - efectos de ramping ... 486

15. Soporte de sonido para sus programas ... 489

15.1 El formato de archivo MOD ... 489
El formato MOD ... 489
El formato 669 ... 496
El formato de archivo Scream Tracker ... 499
El formato S3M ... 502
15.2 Un MOD-Player para la Sound-Blaster ... 509
Principios fundamentales de un MOD-Player ... 511
Mezclar datos de sonido - estructura de un procedimiento de mezcla ... 512
MOD-Player para la tarjeta Sound-Blaster ... 513
Variables necesarias del MOD-Player ... 514
Control de tiempo del MOD-Player - rutinas de temporizador ... 525
Las rutinas de sonido ... 527
Manejo de archivos MOD ... 530
Consejos para la programación de efectos ... 558
El Mod-Player MOD386 ... 563

15.3 Así se programa un MOD-Player para la Gravis Ultrasound ... 569
Estructura del MOD-Player ... 570
Variables importantes del GUS-MOD-Player ... 570
Rutinas centrales del MOD-Player ... 575
El Freak-GUS-Player TCP de «The CoExistence» ... 602

16. Útiles herramientas de Shareware ... 615

16.1 Recupere espacio en el disco - ARJ 2.41 ... 615
El empleo de ARJ ... 615
ARJ - una alternativa a PKZIP? ... 618
16.2 Ahorre espacio con PKLite ... 619
16.3 Rey de los marcadores con Game Wizard ... 621
16.4 Componga con Scream Tracker 3.01b ... 624
El manejo del Scream Tracker ... 625
La composición de un S3M con el Scream Tracker ... 628
16.5 Muestrear con VOC386 ... 629
Las funciones básicas de VOC386 ... 630
Los diferentes efectos de VOC386 ... 632
16.6 Assembler A86 ... 632
16.7 Debugger D86 ... 635
16.8 TOS-Copy, un práctico programa de copia ... 638
16.9 VPIC 6.1 - visualizador gráfico universal ... 639
16.10 PV 2.41 - mostrar y convertir gráficos ... 640
16.11 Convertir gráficos con Image Alchemy 1.7 ... 642
16.12 Ver y convertir gráficos bajo Windows con PaintShop Pro ... 643
16.13 Graphic Workshop for Windows - convertir y visualizar gráficos ... 645
16.14 Calcular geniales gráficos fractales con FractInt 18.2 ... 646
16.15 Calcular gráficos fantásticos con Povray 2.0 ... 647
16.16 Un estupendo entorno para Povray - Moray 1.3 ... 649

Epílogo ... 651

Apéndice ... 653

Contenido del CD adjunto al libro ... 653
Tabla ASCII ... 654

Índice alfabético ... 657

1. Acerca de bytes y phreaking - el lenguaje del mundillo

En el "mundillo" (que es como lo denominamos los que estamos metidos en esto) se ha desarrollado en los últimos tiempos un lenguaje casi imposible de comprender por parte de los profanos. Una conversación entre dos miembros del mundillo podría sonar de la siguiente manera:

Coder a un GFX man:

"¿Cuándo tendrás listo el gráfico para el nuevo anuncio de la BBS?"

Man GFX al Coder:

"Esta noche te lo subo. Conéctate al WHQ. Allí encontrarás un memo con las infos para el code."

Coder al músico:

"¿Qué pasa con la tralla, ya está lista?"

Músico al Coder:

"Te llamaré por voz y luego te haré un upload"

Esto podría ser el extracto de una conversación entre miembros pertenecientes a un grupo de demos como "The CoExistence", "Legend Design", "Xorography" o "KLF". Una persona "normal" sería incapaz de seguir esta conversación, y en la mayoría de los casos extraería conclusiones erróneas sobre el contenido de la misma o tildaría a sus contertulios de locos. Para que esto no le ocurra a usted, explicamos a continuación algunos de los conceptos más importantes:

Concepto	Significado
Intro	Efectos gráficos, a veces con sonido, de un tamaño máximo de 4 Kb
Demo	Efectos gráficos con animaciones, máximo 100 Kb de tamaño.
Mega-Demo	Efectos gráficos con animaciones, tamaño ilimitado
BBS-Addy	Pequeño programa o anuncio publicitario de un buzón de correos, tamaño máximo de 64 Kb
WHQ	World Headquarter (Cuartel general mundial), buzón central mundial de un grupo de demos
...	

Concepto	Significado
...	
EHQ	European Headquarter (Cuartel general europeo), buzón central europeo de un grupo de demos
Tralla	Música
Voz	Llamada telefónica normal, contrario de un módem
Call	La llamada telefónica en sí
Upload, up	Enviar datos
Download, down	Recibir datos
BBS	Bulletin Board System, Buzón de correos electrónico
Coder	Programador
GFX man	Grafista del grupo de demos
Músico	Músico del grupo de demos
Messy, Message	Noticia, correo electrónico
VectorBobs	Objetos gráficos movibles libremente en el espacio tridimensional
ShadeBobs	Pequeños objetos gráficos que proyectan un tipo de sombra sobre la pantalla
Anim	Animaciones, gráficos de todo tipo con movimiento
Scroll	Texto de avance automático en demos e intros

2. El verdadero lenguaje de los programadores del mundillo: ensamblador

A pesar de que actualmente existen lenguajes de programación de alto nivel como Pascal o C y sistemas operativos orientados al usuario como Windows y OS/2, el ensamblador mantiene su vigencia. Este lenguaje dirigido a la máquina permite un acceso directo al corazón (o corazones) del ordenador de una forma que ningún otro lenguaje orientado al problema podría conseguir.

Por esta razón existen ciertas tareas que solamente pueden ser realizadas en ensamblador: Pascal ya empieza a dominar bastante bien la programación de las interrupciones - siempre y cuando no tenga que accederse a los controladores originales para pasar, por ejemplo, pulsaciones del teclado a DOS. Para esta operación no existe ninguna instrucción, por lo que será preciso incluir una parte en ensamblador. También se podría intentar, en Pascal puro, la descarga de un programa residente en memoria, pero aquí también sería necesario utilizar el ensamblador.

Hoy en día, en muchas ocasiones sigue teniendo importancia conseguir un código (Code) lo más compacto posible, por ejemplo durante la programación de programas residentes, ya que se necesitan todos los Kbytes de la memoria principal por debajo de la frontera de los 640 Kb. Por esta razón los lenguajes de alto nivel, con sus bibliotecas de lenguaje que ocupan varios Kbytes, están claramente en desventaja frente a los programas en ensamblador, ya que estos solamente utilizan la memoria que realmente precisan.

La mayor ventaja del lenguaje ensamblador es su velocidad de ejecución. Los lenguajes de alto nivel han conseguido en los últimos años un alto grado de optimización, pero ni la mejor optimización automática puede sustituir la experiencia de un programador. Veamos un ejemplo: si en Pascal se asigna a dos variables seguidas el valor cero, el compilador genera el siguiente código en ensamblador:

```
xor ax, ax
mov var1, ax
xor ax, ax
mov var2, ax
```

Aunque Pascal haya "optimizado" el borrado del registro AX (la posibilidad más rápida es mediante XOR), ha añadido -a causa de las estrictas reglas de compilación antes de la segunda instrucción- otro segundo XOR inútil y superfluo.

Por esta razón no queda otra posibilidad que programar en ensamblador aquellas partes del programa críticas en su duración. Para ello existen dos métodos funda-

mentales: por un lado se puede utilizar el ensamblador interno del Borland Pascal (directiva *asm)*, o recurrir a un ensamblador externo (*TASM, MASM)*, teniendo ambos procesos sus ventajas y desventajas, pero siendo normalmente el ensamblador externo la mejor solución.

Éste está preparado para el ensamblador a causa de su naturaleza y puede aprovechar de mejor manera el lenguaje. Además posee un entorno mucho más potente. Directivas como *db 20 dup* facilitan bastante la vida, pero no pueden ser asumidas por un ensamblador interno. Las macros también son una parte muy importante de los ensambladores externos. Tienen más o menos la misma flexibilidad que los procedimientos, pero su asimilación es mucho más rápida. Por estas razones este libro suele preferir el ensamblador externo. Solamente existe una ocasión en la cual el ensamblador interno tiene las de ganar: cuando una parte de ensamblador tiene que acceder a una variable local desde dentro de un procedimiento (por ejemplo en los procedimientos *GetSprite* y *PutSprite* de la *Unit Sprites*) no se le puede sacar fuera. Aquí esperamos que Borland encuentre alguna solución.

A pesar de la velocidad del ensamblador, esta ventaja se puede perder a causa de una mala programación. Sobre todo los principiantes suelen imitar simplemente las construcciones Basic, lo cual implica una pérdida de velocidad similar a la de la optimización del compilador. Por esta razón, en el siguiente capítulo vamos a dar una pequeña ayuda sobre las partes más importantes de la programación.

2.1 Multiplicaciones y divisiones en Ensamblador

A pesar de la velocidad que se alcanza hoy en día para las multiplicaciones (un 486 DX4 y Pentium consiguen una en sólo 6 ciclos de reloj), deberían seguir usándose en las multiplicaciones y divisiones con potencias de dos las instrucciones de desplazamiento. El número de bits que tiene que ser desplazado corresponde al exponente del multiplicador sobre la base 2, en el caso de una multiplicación por *16* sería por lo tanto 4, dado que 2 elevado a 4 es igual a 16. Cuando se desea multiplicar el registro AX por 8, lo más rápido para conseguirlo es mediante *SHL AX,3*, moviéndose cada bit ocho veces (2 elevado a 3) a una posición superior. Las divisiones se hacen de la misma forma efectuando un desplazamiento hacia la derecha.

Antes se recomendaba utilizar este desplazamiento también en el caso de factores "irregulares". Así, una multiplicación, por ejemplo, por *320* debía ser dividida en una multiplicación por *256* (desplazamiento por 8 bits) y una por *64* (desplazamiento por 6 bits). Luego ya sólo queda sumar ambos resultados (para los matemáticos: división según la ley distributiva). Esto tenía su sentido en los XT, ya que necesitaban para estas multiplicaciones más de 100 ciclos de reloj, pero a partir del 286 este número ha bajado considerablemente (a más o menos 20 ciclos de reloj), de forma que una división de la operación solamente la ralentizaría.

2.2 Cálculos con coma fija

No siempre es posible recurrir a simples variables enteras para efectuar cálculos. Esto ocurre ya cuando se desea dibujar una línea, cuya inclinación casi nunca es de un valor entero. En estos casos se suelen utilizar números reales (Pascal) o número flotantes (C), los cuales se basan en el sistema de números de coma flotante. Estos ofrecen una cantidad determinada de decimales, entre los cuales la coma se puede situar libremente (por esta razón se llama coma flotante). La gran desventaja de estos números reside en su lenta velocidad de proceso. La CPU no posee ningún tipo de operaciones de cálculo sobre una base de hardware. En el caso de que no se posea un coprocesador, cada uno de los cálculos deberá ser efectuado por el software, lo que provoca grandes pérdidas de velocidad de cálculo.

Por el contrario, es más fácil intentar administrar estos números desde Ensamblador. O bien se utilizan las operaciones de lenguajes de alto nivel, lo cual no es nada fácil, dado que, por ejemplo, las operaciones de cálculo elementales bajo Pascal no han sido declaradas *Public*, o hay que escribir sus propias operaciones. Ambas opciones conllevan grandes esfuerzos, por lo que es necesaria una alternativa.

En la mayoría de los casos se puede renunciar a la exagerada exactitud de los números de coma flotante (¿quién necesita once decimales significativos?). La coma flotante tampoco hace falta cuando no se trabaje con grandes cantidades de números. Por lo tanto, la solución se llama: números de coma fija.

Estos números se componen de dos números enteros, en los cuales el primero indica la mantisa y el segundo los decimales, debiendo establecerse el número de éstos. Si la parte anterior a la coma es, por ejemplo, de *17*, y la parte posterior de *1*, en el caso de dos decimales el resultado sería *17,01*, en el caso de un decimal sería *17,1*.

También hay que tener en cuenta la mantisa del número: el valor *-100,3* se divide en el modo de coma fija en los números *-100* y *-0,3* (con un decimal), lo que es bastante lógico, ya que ambos valores deben dar mediante una suma el valor racional deseado. Sumando en este caso el *-100* con el *-0,3* resulta el número *-100,3*, que es lo que se pretendía.

La gran ventaja de estos números está bastante clara: se componen de dos simples números entero enlazados de la forma más simple, sumándose lo que pasa de la parte posterior a la coma de la parte anterior de la misma. Con estos números enteros el procesador también puede trabajar de forma excelente (y rápida), y todo ello sin usar un coprocesador y con un gasto inferior de recursos para los cálculos que en las operaciones de coma flotante.

Lógicamente, estas operaciones deberán ser desarrolladas manualmente, ya que el procesador no está preparado para el cálculo con coma fija. Pero la similitud con los números enteros es tan grande que en principio no causa ningún problema desarrollar nuevas operaciones. También se pueden efectuar funciones complicadas como raíces cuadradas por aproximación, en las cuales se aprecia realmente la ventaja de la velocidad.

Las cuatro operaciones de cálculo básicas

Dado que los números de coma fija son tan similares a los números enteros, no es ningún problema desarrollar de nuevo las cuatro operaciones básicas. El procesador ya dispone de las instrucciones necesarias, y solamente habrá que tener en cuenta que se trata de dos números enlazados.

Para mostrar este sistema añadimos en este capítulo un programa de ejemplo que implementa estas cuatro operaciones fundamentales en Pascal. Lógicamente la velocidad aún se puede incrementar formulando las rutinas en Ensamblador, pero de lo que se trata aquí es de mostrar el método.

Adición

La suma es la operación más simple: se suman de forma separada la parte anterior y posterior a la coma. En estos casos se puede producir un desbordamiento cuando la suma de la partes posteriores a la coma es superior a *1* (en el caso de dos decimales por ejemplo *100*). En este caso el desbordamiento se traspasa a la parte anterior a la coma, incrementando esta en *1* y reduciendo la parte posterior en *1* (en el ejemplo *100*). De forma contraria sucede con los números negativos, en los cuales hay que capturar un desbordamiento por encima de *-1* y reducir la parte anterior a la coma.

Resta

La resta es casi idéntica a la suma, pero en este caso hay que restar los números parciales.

Multiplicación

Para la multiplicación se toma en el siguiente programa un camino distinto, todo ello a efectos demostrativos. Aquí ambos factores se convierten primero en números enteros, se multiplican y luego se reconvierten. Antes de esta conversión hay que dividir por el factor posterior a la coma, ya que éste, durante la conversión a números enteros, se ha multiplicado "demasiado".

División

La división se ha estructurado de forma paralela a la multiplicación. Aquí las versiones de número entero de ambos números se dividen, ampliándose nuevamente con el factor posterior a la coma para mantener la exactitud del cálculo (ya que no existen decimales). Para demostrar este método sirve el programa REGLBASI.PAS.

```
Type Fijo=Record                    {estructura de un nº de coma fija}
          Anterior,
          Posterior:Integer
        End;

Var Var1,                           {variables de ejemplo}
    Var2:Fijo;

Const Max_Post=100;                 {2 decimales}
      Dec_Post=2;

Function Strg(CValor:Fijo):String;
{convierte un nº de coma fija en una cadena}
Var Cad_Post,                       {cadena para formar los decimales}
    Cad_Ant:String;                 {cadena para formar la mantisa}
    i:Word;
Begin
  If CValor.Posterior < 0 Then {parte decimal sin signo}
    CValor.Posterior:=-CValor.Posterior;
  Str(CValor.Posterior:Dec_Post,Cad_Post);
                                    {generar cadena decimal}
  For i:=0 to Dec_Post do           {y sustituir blancos por 0}
    If Cad_Post[i] = ' ' Then Cad_Post[i]:='0';
  Str(CValor.Anterior,Cad_Ant); {generar cadena mantisa}
  Strg:=Cad_Ant+','+Cad_Post;   {componer cadena}
End;

Procedure Convierte(RValor:Real;Var CValor:Fijo);
{convierte a real RValor en nº coma fija CValor}
Begin
  CValor.Anterior:=Trunc(RValor);
    {determinar parte mantisa}
  CValor.Posterior:=Trunc(Round(Frac(RValor)*Max_Post));
    {determinar parte decimal y guardar como nº entero}
End;

Procedure Ajusta(Var CValor:Fijo);
{devuelve el número de coma fija en formato válido}
Begin
  If CValor.Posterior > Max_Post Then Begin
    Dec(CValor.Posterior,Max_Post);  {si decimal rebasa positivo}
    Inc(CValor.Anterior);            {reponer y reducir mantisa}
```

```
    End;
    If CValor.Posterior < -Max_Post Then Begin
      Inc(CValor.Posterior,Max_Post);  {si decimal rebasa negativo}
      Dec(CValor.Anterior);        {reponer y aumentar mantisa}
    End;
  End;

  Procedure Adicion(Var Suma:Fijo;CValor1,CValor2:Fijo);
  {Suma CValor1 y CValor2 y guarda resultado en Suma}
  Var Resultado:Fijo;
  Begin
    Resultado.Posterior:=CValor1.Posterior+CValor2.Posterior;
      {sumar parte decimal}
    Resultado.Anterior:=CValor1.Anterior+CValor2.Anterior;
      {sumar mantisa}
    Ajusta(Resultado);
      {pasar resultado a formato correcto}
    Suma:=Resultado;
  End;

  Procedure Resta(Var Diferencia:Fijo;CValor1,CValor2:Fijo);
  {resta CValor1 de CValor2 y guarda resultado en Diferencia}
  Var Resultado:Fijo;
  Begin
    Resultado.Posterior:=CValor1.Posterior-CValor2.Posterior;
      {restar parte decimal}
    Resultado.Anterior:=CValor1.Anterior-CValor2.Anterior;
      {restar mantisa}
    Ajusta(Resultado);
      {pasar resultado a formato correcto}
    Diferencia:=Resultado;
  End;

  Procedure Multiplicacion(Var Producto:Fijo;CValor1,CValor2:Fijo);
  {multiplica CValor1 y CValor y guarda resultado en Producto}
  Var Resultado:LongInt;
  Begin
    Resultado:=Var1.Anterior*Max_Post + Var1.Posterior;
      {formar primer factor}
    Resultado:=Resultado*(Var2.Anterior*Max_Post+Var2.Posterior);
      {formar segundo factor}
    Resultado:=Resultado div Max_Post;
      {compensar factor aux. Max_Post}
    Producto.Anterior:=Resultado div Max_Post;
      {extraer mantisa y parte decimal}
    Producto.Posterior:=Resultado mod Max_Post;
  End;
```

```
Procedure Division(Var Cociente:Fijo;CValor1,CValor2:Fijo);
{divide CValor1 entre CValor2 y lo guardar en Cociente}
Var Resultado:LongInt;          {resultado intermedio}
Begin
  Resultado:=CValor1.Anterior*Max_Post + CValor1.Posterior;
    {formar contador}
  Resultado:=Resultado*Max_Post div
CValor2.Anterior*Max_Post+CValor2.Posterior);
    {dividir por el divisor, antes disponer de más decimales}
  Cociente.Anterior:=Resultado div Max_Post;
    {extraer parte decimal y mantisa}
  Cociente.Posterior:=Resultado mod Max_Post;
End;

Begin
  WriteLn;
  Convierte(-10.2,Var1);        {cargar dos números de prueba}
  Convierte(25.3,Var2);

  {cálculos propios para demostración:}

  Write(Strg(Var1),'*',Strg(Var2),'= ');
  Multiplicacion(Var1,Var1,Var2);
  WriteLn(Strg(Var1));

  Write(Strg(Var1),'-',Strg(Var2),'= ');
  Resta(Var1,Var1,Var2);
  WriteLn(Strg(Var1));

  Write(Strg(Var1),'/',Strg(Var2),'= ');
  Division(Var1,Var1,Var2);
  WriteLn(Strg(Var1));

  Write(Strg(Var1),'+',Strg(Var2),'= ');
  Adicion(Var1,Var1,Var2);
  WriteLn(Strg(Var1));
End.
```

En este ejemplo se han implementado las operaciones de adición, resta, multiplicación y división en los procedimientos *Adicion, Resta, Mulitplicacion y Division* según los métodos explicados. Como demostración, el programa principal efectúa un pequeño cálculo de cadenas en el cual no aparecen inexactitudes ni después de cuatro pasos aún usando solamente dos decimales.

Las aproximaciones ya nombradas en las sumas y restas se efectúan aquí en un procedimiento llamado *Ajusta*. *Convierte* convierte los números de coma flotante en números de coma fija, y *Strg* genera a partir de este número de coma flotante una cadena, para por ejemplo ofrecer la información por pantalla.

Un ejemplo de aplicación

El programa anterior solamente ha sido una demostración de funcionamiento básico con números de coma fija. Pero para demostrar que este tipo de números también tiene su utilidad, vamos a dar el siguiente ejemplo. En este vamos a desarrollar un logaritmo lineal el cual se acerca bastante en cuanto a velocidad al famoso *Bresenham*. Esto tiene su explicación, ya que este logaritmo de Bresenham en el fondo también trabaja con números de coma fija, aunque no sea tan ostensible.

El procedimiento utilizado se basa en la más simple definición matemática de una recta: $y=mx+b$. Aquí lo más importante es la inclinación m, que indica cuánto asciende la recta en un tramo x de longitud *1*. Pero dado que este valor casi nunca es un número entero, la utilización de los números de coma fija es bastante aconsejable. El procedimiento de ejemplo *Linea* puede dibujar líneas con una inclinación entre *0-1*, para otro tipo de inclinaciones habría que añadir reflexiones etc. (Vea el apartado 8.4).

Este programa utiliza el procedimiento *PutPixel*, descrito en el apartado 4.2. Valga decir aquí que este procedimiento coloca un punto en las coordenadas *(x/y)* en el modo *13h* en el color *Col*. En el CD encontrará este algoritmo de líneas convertido a Ensamblador bajo el nombre de LINEA_FK.PAS.

```
Uses Crt;

Var x:Word;

Procedure PutPixel(x,y,col:word);assembler;
{coloca punto (x/y) en el color col (Modo 13h)}
asm
  mov ax,0a000h          {cargar segmento}
  mov es,ax
  mov ax,320             {Offset = Y*320 + X}
  mul y
  add ax,x
  mov di,ax              {cargar offset}
  mov al,byte ptr col    {cargar color}
  mov es:[di],al         {colocar punto}
End;

Procedure Linea(x1,y1,x2,y2,col:Word);assembler;
asm
  {registros empleados:
    bx/cx: parte decimal y mantisa de la coordenada y
    si   : parte decimal de la inclinación}
  mov si,xl              {cargar x con valor incicial}
```

```
    mov x,si
    sub si,x2              {y formar diferencia x (en si)}

    mov ax,y1              {incializar y (guardado en bx)}
    mov bx,ax
    sub ax,y2              {y formar diferencia y (en ax)}

    mov cx,100             {aumentar diferencia y a causa de precisión}
    imul cx
    idiv si                {y dividir entre x-Diff (inclinación)}
    mov si,ax              {guardar pendiente en si}

    xor cx,cx              {parte decimal de la coord. y a 0}
@lp:
    push x                 {x y parte decimal de y a PutPixel}
    push bx
    push col
    call PutPixel

    add cx,si              {aumentar parte decimal y}
    cmp cx,100             {¿rebase de decimales?}
    jb @no_rebasa          {no, seguir}
    sub cx,100             {si no, reducir parte decimal}
    inc bx                 {y aumentar mantisa}

@no_rebasa:
    inc x                  {incrementar x}
    mov ax,x
    cmp ax,x2              {¿Final?}
    jb @lp                 {no, siguiente pasada}
end;

Begin
    asm mov ax,0013h; int 10h end;{Activar modo 13h}
    Linea(10,10,100,50,1); {dibujar línea}
    ReadLn;
    Textmode(3);
End.
```

El programa principal inicializa a través del BIOS el modo gráfico *13h* y dibuja luego una línea desde las coordenadas (10/10) a (100/50) en el color *Col*. El procedimiento *Linea* aprovecha el hecho de que este algoritmo está limitado a inclinaciones inferiores a 1, de forma que no se precisa la mantisa, y la parte decimal de la inclinación cabe en un registro (aquí *SI*). La coordenada y, que debe tratarse como número decimal, también es colocada en registros -la parte anterior a la coma en *BX* y la parte posterior en *CX*-.

Primero se carga la coordenada x con su valor estándar (x1) y se establece la longitud de la línea en dirección x (x1-x2), procediéndose a continuación de la misma forma con la coordenada y. Ahora se calcula la inclinación: dado que se desea establecer la parte posterior a la coma, primero se multiplica la diferencia y por *100* (dos decimales) y a continuación se divide por la diferencia x, cuyo valor se guarda en *SI*.

El bucle va situando en cada pasada un punto en las coordenadas actuales y calcula posteriormente el siguiente punto. Para ello se incrementa la parte decimal de la coordenada *y* con la mantisa de la inclinación. En el caso de que se produzca aquí un desbordamiento *(>100)* este se traspasa a la mantisa, incrementando ésta en 1 y reduciendo la parte decimal en 100. A continuación se incrementa la coordenada x en 1. Siempre que no se haya alcanzado el punto de destino, el bucle salta al principio.

2.3 Funciones matemáticas propias

Siempre que se utilicen valores con coma flotante se podrá acceder bajo el lenguaje de programación Pascal a una cantidad de funciones suficientemente amplia (no así para los matemáticos, ya lo sabemos). *Seno* y *Coseno*, así como funciones de raíz cuadrada y muchas más, facilitan la programación de problemas matemáticos, pero no la aceleran. Las funciones matemáticas son de lo más lento que se puede encontrar en los lenguajes de programación, a no ser que se pueda recurrir a un coprocesador.

Para la mayoría de tareas de la programación que se efectúa en la práctica, los números enteros son suficiente, siempre y cuando se utilice un rango de valores apropiado (un *seno* de -1 a 1 no tiene mucho sentido en números enteros). Pero si se utilizan las funciones internas de Pascal, este lenguaje no utiliza para ello un procedimiento de números enteros, sino que el número entero se convierte en un número Real y se activa el lento y conocido procedimiento *Real*. Por este motivo, los cálculos con números enteros resultan aún más lentos en Pascal que sus equivalentes con coma flotante. La solución es escribir las funciones uno mismo.

Para ello existen dos variantes fundamentales en la forma de recrear una función: se puede crear una tabla con los valores necesarios al principio del programa, para luego recurrir a ella, o se puede conseguir el valor deseado mediante una aproximación.

Tablas - ¡Que vivan las tablas!

El procedimiento de establecer un valor a través de una tabla seguro que ya es conocido por alguno de los lectores de las clases de matemáticas. Para ello se utiliza una tabla en la cual se pueden buscar los valores deseados. En los casos de determinadas funciones, esto es bastante más rápido que tener que calcular cada valor (sobre todo en el caso de funciones periódicas como el seno). Pero alguien tendrá que haberse dedicado con anterioridad a calcular todos estos valores. Y aquí es exactamente donde cobran sentido las tablas programadas: al arrancar el programa se hace de una vez el trabajo sucio y se genera la tabla con las funciones internas de Pascal, lo que puede tardar varios milisegundos, pero a partir de ahora se podrá recurrir en cualquier momento a estos valores. Como ejemplo vamos a utilizar una función de seno, la cual también se utilizará más adelante en este libro (en la parte gráfica). Por ello, la unidad TOOLS.PAS contiene un procedimiento de uso general (*Sin_Gen*) para el cálculo de la tabla:

```
procedure sin_gen(var tabla:Array of word;per,amp,offset:word);
{calcula una tabla de senos de la longitud per,
 y la deposita en el array tabla.
 Se pide la altura en la variable amp, y la posición del
 punto 0 en offset}

Var i:Word;
Begin
  for i:=0 to per-1 do
    tabla[i]:=round(sin(i*2*pi/per)*amp)+offset;
End;
```

Lo primero que se hace es pasar al procedimiento el nombre del array en el cual se va a crear la tabla, y a continuación la longitud del periodo de la función de Seno. Esta corresponde a la cantidad de registros de la tabla, ya que siempre se calcula exactamente un periodo. La amplitud es ahora el importe del valor más alto. Con una amplitud de 30 la tabla contendrá, por ejemplo, valores de *-30* a *+30*. Como último parámetro se solicita el offset, el cual genera el desplazamiento de la función seno en dirección y. En el ejemplo anterior un offset de *10* generaría una tabla con valores entre *-20* y *40*. El bucle pasa ahora desde el primero al último registro de la tabla y calcula para cada uno de ellos los valores correspondientes con la función de Seno de Pascal. Como ejemplo de aplicación nos servirá un programa para el dibujo de círculos, utilizando para facilitarlo el modo Texto. El siguiente programa de ejemplo, SINTEST.PAS, dibuja dos veces 26 círculos superpuestos, una vez con las funciones normales de seno y coseno y una vez utilizando la tabla. El coprocesador se desactiva al principio, ya que se utiliza poco en la práctica del ensamblador y desvirtuaría bastante el resultado. Arranque una vez el programa y vea la diferencia de ejecución.

```
{$N-}                                   {desactivar coprocesador}
Uses Crt,Tools;

Var phi,                                {ángulo}
    x,y:Word;                           {coordenadas}
    Caracter:Byte;                      {carácter empleado}
    Senos:Array[1..360] of Word;        {acoge la tabla de senos}

Procedure Seno_Real;                    {dibuja un círculo 26 veces}
Begin
  For Caracter:=Ord('A') to Ord('Z')do          {26 pasadas}
    For phi:=1 to 360 do Begin
      x:=Trunc(Round(Sin(phi/180*pi)*20+40));   {calcular coord. x}
      y:=Trunc(Round(Cos(phi/180*pi)*10+12));   {calcular coord. y}
      mem[$b800:y*160+x*2]:=Caracter;           {carácter en pantalla}
    End;
End;
Procedure Seno_Nuevo;                           {dibuja 26 círculos}
Begin
  For Caracter:=Ord('A') to Ord('Z')do          {26 pasadas}
    For phi:=1 to 360 do Begin
      x:=Senos[phi]+40;                         {calcular coord. x}
      If phi<=270 Then                          {calcular coord. y}
        y:=Senos[phi+90] div 2 + 12 Else        {coseno como seno}
                                                {desplazado}
        y:=Senos[phi-270] div 2 + 12;
      mem[$b800:y*160+x*2]:=Caracter;           {carácter en pantalla}
    End;
End;

Begin
  Sin_Gen(Senos,360,20,0);                      {preparar tabla senos}
  ClrScr;                                       {borrar pantalla}
  Seno_real;                                    {dibujar círculos}
  ClrScr;                                       {borrar pantalla}
  Seno_nuevo;                                   {dibujar círculos}
End.
```

El programa calcula primero la tabla de senos. Este proceso solamente se tiene que ejecutar una vez para cada tabla, y a partir de este momento estará siempre disponible. A continuación se activan los dos procedimientos para dibujar los círculos. El primero de ellos, *Seno_Real*, calcula las coordenadas con el ángulo actual utilizando la función de seno y coseno integrada (ambas necesitan el ángulo del arco, por esta razón */180 * pi*). Como radio se utiliza aquí 20 en dirección del eje x y 10 en dirección del eje y y el círculo se coloca en el centro de la imagen mediante las coordenadas *+40,+12*. Al final se coloca el carácter a través de un acceso directo a la memoria de vídeo.

El segundo procedimiento, *Seno_Nuevo* extrae los valores de x e y de la tabla. El coseno se forma mediante un seno desplazado por fases de 90º, teniendo en cuenta los bordes de la tabla. A pesar de la diferenciación *If*, este procedimiento es bastante más rápido que el anterior, lo que queda demostrado ejecutando el programa.

La aproximación

Las tablas sirven de forma excepcional para las funciones con un ámbito de definición concreto, como puede ser el seno, pero las complicaciones empiezan con funciones como la raíz cuadrada, la cual tiene que cubrir un número ilimitado de valores. Si se desea crear un ámbito de definición lo más grande posible, deberá reducirse la resolución de la tabla, y con ello la exactitud de cálculo.

Es por esta razón que encontramos una buena posibilidad para aplicar la fórmula de aproximación. Para la función de raíz cuadrada solamente hay que echar una mirada a un buen libro de matemáticas para encontrar una rápida fórmula de aproximación:

```
Xn+1=1/2 (Xn+a/Xn)
```

Si se aplica esta fórmula para el número *a*, del cual se desea extraer la raíz cuadrada (*Radicando*), y para *Xn* un valor de inicio cualquiera (por ejemplo, 1), se obtiene un valor que está más cerca del resultado exacto. Este valor se vuelve a colocar como *Xn*, obteniéndose un valor más aproximado. Este proceso se puede ir repitiendo hasta que el resultado parezca todo lo exacto que se desea. En el caso de números enteros, esto sucede en cuanto el resultado se diferencia en 0 ó 1 del anterior. Una diferencia de *1* es tolerable, ya que en otro caso el cálculo sería interminable. Esto sucede cuando el resultado acaba saltando entre dos valores vecinos a causa de los redondeos necesarios.

Por cierto: este algoritmo se corrige automáticamente, lo cual tiene su importancia para cálculos "a mano": si el resultado es erróneo, y se utiliza en el siguiente paso como valor de inicio de *Xn*, se efectuará la aproximación a partir de este valor erróneo, lo que alargará el proceso de cálculo, obteniéndose a pesar de ello el resultado correcto.

Este algoritmo se encuentra en el procedimiento de Ensamblador *Raiz* dentro del archivo RAIZ.ASM. Este procedimiento no se ha colocado en una unidad ya que se precisará más adelante (en los gráficos en 3-D) como procedimiento *Near*. Un procedimiento *Far*, como los que genera una unidad, sería demasiado lento al llamarlo. El texto en ensamblador contiene dos procedimientos: por un lado *Raíz*, el cual contiene el cálculo en sí. Este procedimiento trabaja de forma orientada al registro, lo que significa que el traspaso de parámetros se efectúa a través de los registros *DX:AX*. La aplicación 3-D accede directamente a este procedimiento.

Para poder calcular directamente desde Pascal la raíz cuadrada en aquellos casos en los cuales la velocidad no sea tan importante, el archivo contiene otra función "marco" (*Func_Raiz*), a la cual se pasa de forma normal el radicando como parámetro y la cual, después de activar *Raiz*, devuelve como resultado de la función el valor de la raíz cuadrada.

```
.286                          ;activar comandos 286
e equ db 66h                  ;Operand Size Prefix (instrucc. 32-bits)
w equ word ptr

code segment public
assume cs:code
public raiz
public func_raiz
                              ;valor radicando en dx:ax
raiz proc pascal              ;resultado en ax (Function)
e                             ;cálculo en 32 bits
   xor si,si                  ;borrar resultado intermedio (en esi)
   db 66h,0fh,0ach,0d3h,10h ;shrd ebx,edx,16d - dx a ebx (16 bits sup.)
   mov bx,ax                  ;ax a ebx (abajo) - dx:ax ahora en ebx
e
   xor dx,dx                  ;borrar edx
e
   mov cx,bx                  ;guardar valor inicial en ecx
e
   mov ax,bx                  ;cargar eax
repetir:
e
   idiv bx                    ;dividir entre Xn
e
   xor dx,dx                  ;Resto no interesa
e
   add ax,bx                  ;sumar Xn
e
   shr ax,1                   ;dividir por 2
e
   sub si,ax                  ;diferencia al resultado anterior
e
   cmp si,1                   ;<= 1
   jbe final                  ;listo
e
   mov si,ax                  ;guardar resultado como anterior
e
   mov bx,ax                  ;marcar como Xn
e
   mov ax,cx                  ;cargar de nuevo valor inic.
   jmp repetir                ;y al inicio del bucle
```

```
final:
  ret                          ;resultado ahora en eax
raiz endp

func_raiz proc pascal a:dword  ;convierte proced. en función Pascal
   mov ax,word ptr a           ;escribir parámetro en registro
   mov dx,word ptr a+2
   call raiz                   ;y sacar raíz
   ret
func_raiz endp

code ends
end
```

En el procedimiento *Raiz* resaltan a primera vista la gran cantidad de "e" al principio de las líneas, las cuales significan *db 66h* , tal como se declara al principio. Se trata del *Operand-Size-Prefix* del 386, el cual amplía la siguiente instrucción a 32 bits. Las instrucciones de 32 bits consiguen aquí un gran incremento de la velocidad, dado que los resultados *LongInt* no tienen que ser divididos entre dos registros. Por desgracia, el Pascal aún no es capaz de evaluar estas instrucciones directamente, aunque procedan de un ensamblador externo. Por esta razón solamente queda la posibilidad de activar de forma manual cada instrucción a 32 bits.

Primero se desplaza el contenido del registro *DX* mediante la instrucción del 386 *shrd* a la parte superior del *EBX*, y después la parte inferior con *Ax*. *ECX* actúa aquí como memoria para el radicando *a*, el cual también se necesita más adelante.

El siguiente bucle de iteración efectúa los cálculos descritos en la fórmula: después de la división del radicando por el último valor de aproximación (en *EBX*), este aún se suma al cociente, cerrándose de esta forma los paréntesis. Finalmente se divide por 2 y se compara el resultado con el anterior. En el momento de coincidir (desviación máxima de 1) se finaliza, de lo contrario, se carga *Xn (en BX)* con el resultado actual y se continúa la iteración. Al final, la raíz cuadrada se encuentra en el registro *AX*, donde también coloca el resultado una función Pascal. Como ejemplo sirva también aquí una comparación de velocidad:

```
{$n-}                          {desactivar coprocesador}
Function Func_raiz(Radicando:LongInt):Integer;external;
{$l raiz}
{¡aquí se ha de indicar la vía de acceso del módulo RAIZ.OBJ!}

var i:word;                    {contador de bucle}
    n:Integer;                 {resultado de cálculo entero}
    r:Real;                    {resultado de cálculo real}
Procedure Raiz_nueva;          {calcula raíz con aprox. entera}
```

```
Begin
  For i:=1 to 10000 do              {recorrer 10.000 veces}
    n:=Func_raiz(87654321);         {para obtener comp. velocidad}
End;

Procedure Raiz_real;                {calcula raíz mediante función
Pascal}
Begin
  For i:=1 to 10000 do              {recorrer 10.000 veces}
    r:=Sqrt(87654321);              {para obtener comp. velocidad}
End;

Begin
  writeLn;
  WriteLn('Cálculo de raíz mediante Pascal - función comienza');
  Raiz_real;
  WriteLn('Resultado: ',r:0:0);
  WriteLn('Cálculo de raíz mediante enteros - función comienza');
  Raiz_nueva;
  WriteLn('Resultado: ',n);
End.
```

Este programa, llamado RAIZTEST.PAS, calcula de dos formas diferentes 10000 veces la raíz cuadrada de 87654321. Después de arrancar, hasta un 486 (con el coprocesador desactivado, opción de compilador $n) tarda unos segundos en ofrecer el resultado. La segunda parte (cálculo propio) se ejecuta en fracciones de segundo.

2.4 Optimizar comparaciones

Las comparaciones son, después de las operaciones de cálculo, las tareas más difíciles para un procesador, por lo cual deberán usarse poco y optimizarse lo máximo que se pueda.

OR en vez de CMP

Una forma de aceleración simple la ofrecen los operadores lógicos del procesador. Entre estos está la instrucción *Test*, la cual, en el fondo, utiliza una instrucción *AND*. Esta instrucción es apropiada cuando se quieren comprobar combinaciones específicas de bit. Aún se puede conseguir una mayor optimización en comparaciones con *0* utilizando la instrucción *OR*. *OR AL,AL* ofrece con un *0* en *AL* un Flag a cero (a bifurcar con *jz,jnz*), en otro caso uno borrado.

Dado que esta instrucción sitúa todos los Flags a los valores correspondientes al contenido del registro, también se puede comprobar con ello el prefijo del número: La instrucción *js* salta en el caso de un signo negativo a la dirección especificada.

Comparaciones con desbordamiento

En el caso de comprobaciones sobre la exactitud de coordenadas, se precisa en muchos casos la comparación en ambas partes. Por un lado, el valor no debe ser inferior a *0*, y, por otro, no debe superar *320*. Aplicar aquí dos comparaciones separadas sería una pérdida de tiempo. Es mucho más elegante efectuar una comparación sin prefijo. Si se efectúa una comprobación de la coordenada sin prefijo a un valor inferior a *320*, también se incluirán valores negativos, ya que estos tienen valores superiores a *32767* si se comparan sin prefijo, con lo cual cumplen la condición.

Comparaciones de cadenas

Para muchos, las comparaciones de cadenas siguen siendo un misterio. Pero este tipo de comparaciones son de importancia fundamental para, por ejemplo, la programación de un TSR que tiene que comprobar si ya está presente en la memoria. Para comprender esta comparación es muy importante el funcionamiento de la instrucción *JCXZ*. Esta instrucción salta a la dirección de memoria indicada si el registro *CX* es igual a 0. En combinación con la instrucción de repetición *REPE CMPSB* se puede programar de manera fácil una comparación de cadenas.

Primero se cargan los dos punteros *ES:DI* y *DS:DI* sobre las dos cadenas y se guarda la longitud de la misma en *CX*. A continuación se ejecuta la instrucción *REPE CMPSB*, la cual irá comparando hasta que las cadenas presenten diferencias (*Zero Flag* borrado, REPE se interrumpe) o haya finalizado la cadena (*CX=0*, o sea, condición final de un bucle de repetición). La siguiente instrucción *JCXZ* se apropia de esta diferencia. Si las cadenas son diferentes, *CX* aún no es 0, o sea, que no se produce ningún salto. En cuanto las cadenas sean completamente iguales *CX* será igual a 0 y se bifurcará.

2.5 Variables en ensamblador

Siempre se debería intentar mantener la mayor cantidad posible de valores en los registros, ya que el procesador puede acceder a ellos de una forma mucho más rápida y directa. Para ello no se deberá dudar nunca en utilizar registros con tareas específicas (*SI, DI, BP)* a modo de variables de memoria normales. Pero a pesar del máximo aprovechamiento del juego de registros, nunca se tienen suficientes (una deducción de la ley de Murphy: Todo lo que puede salir mal, sale mal) y hay que hacer uso de la memoria convencional, debiéndose utilizar para ello las variables normales.

Acceso a variables de Pascal

Actualmente, el acceso a variables de Pascal desde ensamblador ya no es ningún problema, aunque sigan existiendo programadores que se complican la vida con construcciones como *mov AX, [offset variable]*. Es mucho más simple utilizar directamente *mov ax, variable*. Si se desea realizar al mismo tiempo una conversión de tipo (por ejemplo, Offset de puntero a registro de 16 bits) habrá que incluir un *Word* o *Byte Ptr: move AX, word ptr Puntero+2*.

Acceso a arrays y records

Desde ensamblador también se puede acceder directamente a los arrays, lo único que hace falta es encargarse uno mismo de la indexación. Para ello hay que tener en cuenta sobre todo el tamaño de los elementos: en el caso de registros *word* los elementos individuales tienen una separación de 2 *byte*, en el caso de *Doubleword* una separación de 4 *byte*. También son posibles otros offset, por ejemplo en el caso de un *Array of Record*. El 386 ya fue preparado para ello, y así es capaz de acceder a variables en la forma *mov AX,[2*ecx]* . Esto es bastante difícil de realizar en Pascal (solamente tiene código 286), ya que cada instrucción de este tipo debería ser colocada completamente como cadena de bytes. Por esta razón es mejor efectuar la multiplicación externamente mediante un comando *shl SI1*.

En la mayoría de ensambladores tampoco se puede indicar el offset del array de la forma usual antes del índice: la instrucción *mov AX, word ptr ARR(SI)* es convertida por ensamblador en *mov AX, word ptr (SI+offset Arr]*.

Los registros tampoco tienen que ser indicados hoy en día de forma complicada a través de offsets constantes. Desde Pascal ASM se puede acceder al igual que desde Pascal directamente a los registros: *mov AX, word ptr rec.a*. También el TASM y el MASM poseen una variable similar a un registro (*record*), la estructura. Aquí se recrea un registro *(record)* Pascal para poder acceder a él como si fuera desde Pascal:

```
data segment

rec_tipo struc
  a dw ?
  b db ?
rec_tipo ends

extrn rec:rec_tipo
```

Variables de segmento de código (codesegment)

Por desgracia, el programador no suele tener suficientes registros de procesador a su disposición, por lo cual se alegra por cada uno que gane adicionalmente. Aquí se puede aprovechar el registro *BP*, al cual se puede acceder libremente de la misma forma que a todos los demás registros. Pero el asunto tiene un pequeño inconveniente: el *BP* se utiliza para el direccionamiento de las variables locales del procedimiento. Si se utiliza el registro *BP* o bien se renuncia a estas variables o hay que trasladarlas a otro lugar. Las variables globales normalmente también son inaccesibles, sobre todo en el caso de procedimientos gráficos, ya que el registro *DS* deja de señalar al segmento de datos, y cambia por ejemplo a los datos de los Sprites. En estos casos, la única posibilidad disponible son las variables de segmento de código.

Estas se encuentran junto con el código del programa en el segmento de código actual y su direccionamiento se efectúa a nivel de lenguaje máquina a través del prefijo override del segmento. A nivel de programador esta tarea es asumida por Ensamblador, de manera que no se nota diferencia alguna. Lo único que no es posible es el acceso desde el exterior, por ejemplo, desde otros procedimientos o módulos, pero lo mismo ocurre con las variables locales. Bajo TASM y MASM se pueden crear directamente las variables dentro del segmento del código en vez del segmento de datos, encargándose el Ensamblador de forma automática del direccionamiento correcto.

El lenguaje Pascal complica las cosas un poco más, ya que no es posible ocupar el segmento de código con datos fuera de los procedimientos o funciones. Por esta razón hay que emplear aquí un pequeño truco: Al principio del procedimiento se sitúa una instrucción *Jmp* al código de procedimiento, e inmediatamente después se crean las variables en etiquetas (labels). Esto quedaría más o menos así:

```
Procedure test;ensamblador;
asm
   jmp @los

@Var1: dw 0
@Var2: db 0

@los:
   {resto del procedimiento}
End;
```

Aquí hay que insertar, en cada acceso a variables, un *word ptr* o *byte ptr*, ya que para Pascal se trata, en los casos de @Var1 y @Var2, de etiquetas y no de variables.

Arrays circulares

Con los arrays no siempre se trabaja de adelante hacia atrás, sino que en muchas ocasiones se hace en forma circular, del principio al fin y otra vez al principio. Esto se suele aplicar, por ejemplo, en el caso de tablas senoidales, a fin de poder abarcar también ángulos de 700 grados o similares. La forma más simple, pero también la más lenta, se soluciona mediante consultas, las cuales proyectan el índice al llegar al final del array de nuevo hacia el interior del array. En el ejemplo se restarían *360* grados, de forma que se utilizaría *340* como índice.

Una forma mucho más rápida para realizar esto es estructurar el array de tal manera que la cantidad de registros correspondan a una potencia de 2, o sea *32, 64, 128,* etc. En estos casos el índice del array se puede proyectar hacia el array de forma simple mediante un enmascaramiento del bit con un AND. Si el array tiene 64 registros (0-63) se sitúa en cada índice un *63 geANDet*, mediante lo cual se ocultan los dos bits superiores y solamente cuentan los seis bits inferiores.

Para estructurar un array de esta forma hará falta el número correcto de elementos. Para ello se puede indicar por ejemplo durante la generación del seno un periodo de *64*. No se obtendrán valores de grado enteros, pero esto carece de importancia. ¿Quién dice que no se puede dividir un círculo en *64* en vez de en *360* partes?

Enmascaramiento rotatorio de bits

En la programación de sistema se suelen utilizar enmascaramientos de bits. Para ello se escribe un determinado valor en un registro específico, por ejemplo el registro de la VGA, teniendo cada bit una tarea diferente, como por ejemplo la activación de los correspondientes planos de bit. Para seguir con nuestro ejemplo: suele ser preciso seleccionar uno tras otro los planos *0,1,2,3* y luego otra vez *0*, situando de forma correspondiente los *bits 0,1,2,3* y después otra vez el *0*.

Pero se produce el problema de tener que volver después del *bit 3* de nuevo al *bit 0*. Un método poco elegante consiste en una instrucción *CMP*, pero existe una manera mucho más rápida: a nivel de 8 bits se generaría para ello una rotación de bits en la cual el *bit 7* se traslada automáticamente al *bit 0*. Al nivel de 4 bits que se precisa aquí se puede trabajar de la misma forma, pero, dado que no es posible efectuar una rotación de medio Byte, hay que utilizar un pequeño truco: Al principio se carga el Byte con *11h* en vez de *01h*, obteniendo la parte superior e inferior del bit (bit-nibble) el mismo valor. Si ahora se efectúa una rotación y se utiliza para el enmascaramiento el nibble inferior, se conseguirá el efecto normal con un *enmascaramiento 0-3*. Después de esto el byte tendrá un contenido de *88h*. Si se vuelve a rotar este valor, volverá a aparecer como siguiente el valor *11h (bit 7 a bit 0, y bit 3 a bit 4)*.

Enmascaramiento de un número determinado de bits

En algunos casos es preciso seleccionar un número determinado de bits dentro de un WORD o Byte, por ejemplo en el cargador GIF del apartado 4.3. Hay que ir cogiendo un número de bits no evaluado del byte anterior. En este caso se puede utilizar un camino aritmético en vez de una complicada programación de bucles: se carga un registro (por ejemplo *AX)* con *1* y se desplaza este registro hacia la izquierda por el número de bytes que se desean enmascarar. Ahora se reduce este valor en *1* y se obtendrá la máscara en la que todos los bits necesarios contienen un *1* y todos los demás un *0*. Como fórmula sería así:

```
Enmascaramiento := (1 shl cantidad) - 1
```

Si se desea seleccionar, por ejemplo, *6 bits* esto daría como resultado la máscara *(1 shl 6)-1 = 63*, colocándose los bits 0-5 y borrándose los bits 6 y 7. Ahora solamente hará falta añadir con un *AND* el byte que se desea enmascarar con este valor, y se obtendrán los bits buscados.

En relación con ello existe una curiosa característica de las instrucciones *SHR y SHL* a partir del 386. Cuando se desplazan bits, éste solamente tiene en cuenta los 5 bits inferiores. De esta forma el máximo desplazamiento posible es de 31 bits, independientemente del ancho del registro. Si se desea desplazar, por ejemplo, el registro *AX* por 34 bits, lo que equivaldría a un borrado (ya que AX solamente tiene un ancho de 16 bits), ejecutando un *SHL AX,34,d*, en realidad solamente se conseguirá un desplazamiento de 2 bits. Esto no suele tener importancia, pero nos ha hecho perder en algunos casos bastante tiempo buscando el error, ya que contábamos que utilizando un valor de desplazamiento alto se borraría el registro.

2.6 El secreto de los cracks - las interrupciones

A pesar del interés y versatilidad de las interrupciones y su programación para muchos siguen siendo un misterio. La causa suelen ser las continuas caídas del sistema que se pueden producir de esta forma. Pero teniendo los conocimientos básicos necesarios y con la ayuda de algunos ejemplos, se puede aprender rápidamente la segura utilización de las interrupciones. Si a pesar de ello su sistema suele "caer", no se preocupe, ya que es algo que le pasa hasta al más experto programador de interrupciones.

Las interrupciones son primordialmente órdenes de interrupción que llegan de una forma u otra al procesador; se diferencia entre interrupciones de software y de hardware. Las primeras se ejecutan de forma explícita con la instrucción *Int* y son comparables a simples subprogramas. La segunda categoría es bastante más

interesante: las interrupciones del hardware provienen de periféricos externos y llegan a la CPU a través de los dos controladores de interrupciones. Así, como ejemplo, cada pulsación de una tecla provoca una interrupción que hace que el procesador ejecute un programa (controlador de interrupción) que obtiene el carácter del teclado.

Desviar vectores

Aquí comienza el trabajo del programador. Puede añadir a cada interrupción su propio programa que ejecute determinadas acciones, como, por ejemplo, en el caso de la interrupción del teclado, la emisión de un pitido a través del altavoz con cada pulsación de una tecla. La dirección del procedimiento a ejecutar para cada interrupción se guarda en vectores de interrupción al principio de la RAM. Este vector puede ser leído y escrito mediante las funciones DOS *35h* y *25h*. A las funciones se les transmite en el registro *AL* el número de la interrupción. La función *35h* devuelve en *ES:BX* el vector, mientras que en la función *25h* hay que incluirlo en los registros *DS:DX*. Si se desea leer, por ejemplo, la interrupción 9, se haría de la siguiente forma:

```
mov ax, 3509h          ; número de función e interrupción
int 21h                ; Ejecutar la función DOS
```

Ahora el vector está situado en es:bx. Se coloca mediante las siguientes instrucciones:

```
lds dx, Vector         ; Extraer el vector (como puntero)
mox ax, 2509h          ; número de función e interrupción
int 21h                ; Ejecutar la función DOS
```

Antes de desviar un vector a su propio procedimiento deberá leer y guardar el vector anterior. Al final del programa, o en el caso de TSR cuando se descargue, lo necesitará para restablecer la situación inicial. Otra aplicación es la llamada al controlador original a partir del propio.

Llamada al controlador original y finalización

En el caso del clic del teclado tendría poco sentido emitir solamente un pitido con cada pulsación de una tecla, ya que esta tecla también debería aparecer como carácter en la pantalla. Para ello será necesario ejecutar el controlador antiguo antes o después de la emisión del pitido, a no ser que desee escribir su propio controlador de teclado. Ya que se ha guardado el vector de interrupción antiguo, se puede utilizar casi directamente como destino de un salto en un *Far Call. Casi directamente*, ya que hay que simular una llamada a una interrupción. La única diferencia entre una llamada a una interrupción frente a una instrucción usual *Far Call* consiste en el almacenamiento del Flag

del procesador, lo que se puede recrear de forma simple mediante una instrucción *pushf*. Una llamada completa tendría la siguiente estructura:

```
pushf
call dword ptr (VectorAnterior)
```

donde el vector original debe estar guardado en el puntero *VectorAnterior*. Una interrupción se finaliza con la instrucción *IRET*. Antes habría que restablecer, en cualquier caso, el estado original del registro del procesador. También es posible que la interrupción se activara en medio de una rutina que precise ciertos registros.

Bloqueo de interrupciones

Para el bloqueo de todas las interrupciones a la vez el procesador posee su propia instrucción *CLI*, que cierra la entrada a las interrupciones, volviéndose a abrir con la instrucción *STI*. Pero en algunos casos solamente se precisa bloquear algunas interrupciones, manteniendo otras activas. Para ello habrá que cambiar el programa de los controladores de interrupciones. Estos controladores utilizan una forma de contar un poco diferente a la de los vectores: las interrupciones de Hardware tienen los números *0-7* (Controlador 1) y *8-15* (Controlador 2). Aquí se habla de los IRQs (Interupt-Request), refiriéndose la denominación "interrupción" al número del vector.

Los *IRQs 0-7* se anuncian a la CPU por parte del Controlador 1 como interrupciones *8-0fh*, los *IRQs 8-15* por el controlador 2 como interrupciones *70h-77h*. Ambos controladores están conectados a través del *IRQ 2* (en forma de cascada), lo que significa que si el controlador 1 recibe una instrucción de Interrupción traspasará esta a través de la entrada 2 al controlador 2. La ocupación de los controladores es como sigue:

Controlador 1:	
IRQ	**Propietario**
0	Reloj
1	Teclado
2	enlazado en forma de cascada con el controlador 2
3	Com 2, Com 4
4	Com 1, Com 3
5	LPT 2
6	Disquete
7	LPT 1

Controlador 2:	
IRQ	Propietario
8	Reloj a tiempo real
9	VGA (muchas veces inactivo), red
a	-
b	-
c	-
d	Coprocesador
e	Disco duro

Las interrupciones están ordenadas jerárquicamente, lo que significa que los IRQs con números inferiores tienen una prioridad superior y por lo tanto preferencia. Esta instrucción se puede variar reprogramando el controlador. Pero es preferible no hacerlo, ya que el BIOS y el DOS se basan en esta estructura.

Los controladores poseen los llamados IMRs (Interupt Mask Register), mediante los cuales es bastante fácil ocultar determinadas interrupciones. El IMR del primer controlador se encuentra en la dirección de puerto *21h*, el del segundo en el puerto *a1h*, donde un bit colocado significa una interrupción bloqueada. Para desconectar, por ejemplo, el reloj a tiempo real, se utilizarían las siguientes instrucciones:

```
in al,0alh        ;cargar el IMR2
or al,01h         ;situar el bit 0
out 0a1h,al       ;y volver a escribir
```

Ambos controladores poseen otra segunda dirección de puerto en *20h* y *a0h*, a través de las cuales se pueden enviar instrucciones. La instrucción más importante es la instrucción *EOI* (End of Interrupt) con el número *20h*, que muestra el final del controlador de interrupción. Si se llama a los controladores originales desde el controlador propio, este se encargará de esta tarea. Pero cuando se utilice un controlador completamente nuevo habrá que encargarse de que al final del controlador se envíe una instrucción *20h* a los puertos *20h* o *a0h*:

```
mov al, 20h
out 20h, al
```

Retorno al DOS - Reentrance

En dependencia de la aplicación, un controlador de interrupción puede durar pocos ciclos de procesador (los clics del teclado) o varios segundos (p.ej. Imprimir pantalla, *Int 5*). En el caso de estos últimos habrá que evitar que durante la ejecu-

ción del controlador este vuelva a procesarse mediante otra interrupción. En el mejor de los casos se produciría un resultado no esperado (en el caso de la impresión de pantalla esto produciría dos impresiones anidadas), pero lo que se produce normalmente es una caída del sistema. La razón de ello es que el nuevo controlador accede a variables que el controlador interrumpido aún precisa. La mejor posibilidad para evitar esto es utilizando una variable *Flag* en la cual se registra que el controlador se ha arrancado y a la que se puede recurrir antes de una nueva activación, a fin de evitar una reentrance.

Un caso más complicado de reentrance son las TSR de mayor tamaño que activan mediante una pulsación de una tecla programas completos con menús. En este caso, el problema reside en el DOS, el cual no permite la ejecución casi simultánea de varias funciones DOS. Cuando una interrupción interrumpe una instrucción DOS y arranca otra función DOS durante su ejecución (por ejemplo para la representación en pantalla), esta llamada destruirá el stack del DOS, de forma que después de procesarse el controlador y volver a la función DOS interrumpida el sistema se caerá.

Evitar esto no es tan simple: primero hay que consultar el flag llamado *InDos*. Su posición en memoria se puede averiguar antes de la instalación del handler mediante la función no documentada del DOS *34h*, la cual devuelve un puntero *Far* a los registros *ES:BX*. Solamente se podrá bifurcar dentro de un controlador a funciones DOS cuando este flag contenga un *0* al activarse el controlador. Además, se debería instalar un controlador para la interrupción *28h*, la cual es llamada siempre que el COMMAND.COM esté esperando una introducción del usuario en la línea de comando. En este caso el flag *InDos* puede ser *1*, ya que el COMMAND.COM en sí cuenta como una función DOS. Estas complicadas medidas de seguridad se pueden evitar siempre que el controlador no arranque funciones DOS, lo cual no representa ningún problema para la mayoría de los TSR.

La captura de CTRL-C y Reset

En los programas profesionales, en teoría, debería ser imposible salir por "una pequeña puerta trasera". Porque en el caso de que aún hubiese archivos abiertos, la consecuencia directa sería la pérdida de datos. Por otra parte, en las demostraciones no queda bien que cualquier espectador pueda interrumpir la presentación mediante un Ctrl-Break o un Ctrl+Alt+Supr. Pero ¿cuál es la mejor manera de capturar estas funciones del BIOS?

Para las funciones Ctrl+C y Ctrl+Break el DOS ya ofrece un camino bastante seguro: al pulsar una de estas teclas el DOS llama al Interrupción *23h*, el cual provoca la caída del sistema. Lo que se puede hacer simplemente es desviar este vector a una rutina propia, la cual solamente vuelve al llamador. La desventaja es que no suele funcionar, sobre todo con el Ctrl+Break, lo que en ciertas circunstancias provoca una caída del sistema.

Existe un método mucho más seguro, el cual también captura el Ctrl+Alt+Supr: dentro de un controlador propio de la interrupción del teclado se comprueba antes de la llamada al controlador original si se ha pulsado alguna de las combinaciones de teclas críticas. En este caso se finaliza el controlador, y no sucede absolutamente nada. Las teclas legales se transmiten al controlador antiguo, el cual las envía al programa principal. Para demostrar este método encontrará en el CD el programa NO_RST.ASM, el cual tiene que ser ensamblado con TASM o MASM a un archivo EXE.

```
data segment public
mensaje_inicio: db 'Ya no es posible un reset',0dh,0ah,'$'
buffer:     db 40d                  ;longitud del búfer de entrada
            db 40 dup (0)           ;búfer
old_int9    dd 0                    ;controlador de interr. antiguo
data ends

code segment public
assume cs:code,ds:data
handler9 proc near                  ;nuevo controlador de interr. 9
  push ax                           ;guardar registros empleados
  push bx
  push ds
  push es
  mov ax,data                       ;cargar ds
  mov ds,ax

  in al,60h                         ;leer caracteres del teclado a al

  xor bx,bx                         ;es al segmento 0
  mov es,bx
  mov bl,byte ptr es:[417h]         ;cargar estado de teclado en bl

  cmp al,83d                        ;código scan de la tecla Supr?
  jne no_reset                      ;no, no es reset

  and bl,0ch                        ;enmascarar Ctrl y Alt
  cmp bl,0ch                        ;¿ambas pulsadas?
  jne no_reset                      ;no, no es Reset

Block:                              ;Reset o Break, bloquear
  mov al,20h                        ;EoI al Interrupt-Controller
  out 20h,al
  jmp terminado                     ;y abandonar interr.

no_reset:                           ;no es Reset, comprobar Break
  cmp al,224d                       ;¿tecla ampliada?
  je evtl_Break                     ;si -> posible Break
  cmp al,46d                        ;Tecla 'C' ?
  jne legal                         ;no -> tecla legal
```

```
evtl_Break:
  test bl,4                        ;comprobar estado tecla Ctrl
  jne Block                        ;pulsada, bloquear


legal:                             ;tecla legal -> llamar contr. ant.
  pushf
  call dword ptr [old_int9]        ;llamar control. original
terminado:
  pop es
  pop ds                           ;recuperar registros
  pop bx
  pop ax
  iret
handler9 endp

start proc near
  mov ax,data                      ;cargar ds
  mov ds,ax
  mov dx,offset mensaje_inicio     ;cargar dx con offset del mensaje
  mov ah,09h                       ;visualizar mensaje
  int 21h

  mov ax,3509h                     ;leer antiguo vector de interr.
  int 21h
  mov word ptr old_int9,bx         ;y guardar
  mov word ptr old_int9 + 2, es

  push ds                          ;guardar ds
  mov ax,cs                        ;cargar con cs
  mov ds,ax
  mov dx,offset handler9           ;cargar también offset de control.
  mov ax,2509h                     ;fijar vector
  int 21h
  pop ds

;-----------------------------------------------------------------------
;Aquí también puede hacer una llamada a su programa en vez del DOS

  mov ah,0ah                       ;leer cadena de caracteres
  lea dx,buffer                    ;como programa de ejemplo
  int 21h
;-----------------------------------------------------------------------

  push ds
  lds dx,old_int9                  ;restaurar vector antiguo
  mov ax,2509h
  int 21h
  pop ds
```

```
  mov ax,4c00h                    ;terminar programa
  int 21h
start endp

code ends
end start
```

El programa principal (*start*) devuelve primero una cadena explicativa, lee el vector anterior de la interrupción del teclado *9* y sitúa el nuevo vector en el procedimiento *Handler9*. A continuación se llama a la introducción de caracteres del DOS, en sustitución de la parte del programa protegida (por ejemplo las rutinas de demo), que aquí recibe una cadena de 40 caracteres. Finalmente se restaura el controlador anterior y finaliza el programa.

El controlador en sí es activado con cada pulsación de una tecla. Primero asegura todos los registros ocupados para que el programa interrumpido no note nada de las actividades del controlador. A continuación se establece a través del puerto de datos del controlador del teclado el código de la tecla pulsada (situado en *AL*), y se lee el estado de las teclas Ctrl y Alt a través de la variable de estado del teclado (situada en *BL*). Esta variable se encuentra en la *dirección 0:417h* y tiene la siguiente ocupación:

Bit	Significado
7	INS
6	BLOQ MAYUS
5	BLOQ NUM
4	BLOQ DESPL
3	ALT
2	CTRL
1	SHIFT (mayúscula) IZQUIERDO
0	SHIFT (mayúscula) DERECHO

Primero se comprueba si se ha pulsado la tecla Supr (indica un RESET), y a continuación si están pulsadas Ctrl y Alt (*bit 2 y 3 en BL*). En el caso de que se den ambos factores se continúa en la etiqueta *Block*. En este punto se envía simplemente una señal *EOI* al controlador de la interrupción 1 y se salta al final de controlador. En el caso de que no se haya efectuado ningún reset, se comprueban a partir de la etiqueta *no_reset* las combinaciones Ctrl+Break y Ctrl+C. Si no se ha pulsado la tecla C (código 46) ni una tecla ampliada (código 224) se podrá deducir que se ha pulsado una tecla normal (legal), y se llama al controlador antiguo en la etiqueta *legal*. Si se ha pulsado Break o C se comprueba también la tecla Ctrl. Si está activada, se bloquea, en caso contrario se tratará de una tecla legal. Si desea utilizar esto en sus propios programas, deberá activar su propio procedimiento principal en vez de la entrada de caracteres del DOS.

2.7 Consejos para la programación de bucles

Hasta en los temas tan simples como la programación de bucles existen en el lenguaje máquina bastantes posibilidades de optimización. Esto comienza con la construcción típica de una bucle simple: *loop label*. Al parecer esta instrucción fue dejada de lado por los desarrolladores de la CPU a lo largo de los últimos tiempos. En los 386 una construcción cómo *dec CX, jne label* es casi un 10% más rápida, en un 486 esto llega al 40%! Y todo ello a pesar que con esta última instrucción se tiene que extraer una byte más de la memoria principal. La instrucción de bucle solamente tiene sentido cuando el decremento de *CX* no puede influir en los flags, por ejemplo en el caso de complicadas comparaciones de cadenas que no pueden solucionarse con *REP CMPSB*.

El Flag de dirección en instrucciones de cadena (string)

Una fuente de error bastante común en las instrucciones de cadena (*lodsb, cmpsb,* etc.), que en el fondo también son bucles, suele ser según nuestras experiencias el Flag de dirección (*Direction Flag*), el cual indica la dirección en la que se procesará la cadena. Este Flag suele estar borrado, pero si lo va a utilizar en alguna parte de su programa para comprobar una cadena de atrás hacia adelante, no se olvide nunca de volver a borrarlo!

Anidar

En el caso de bucles anidados habrá que buscar siempre una solución de compromiso entre la velocidad y la utilización de registros. Debería usarse el máximo número posible de registros como contadores de bucle, debiéndose recurrir posteriormente a variables de memoria. La cuenta de los bucles se puede manipular mediante una correcta selección de los límites, así debería contarse siempre desde atrás, a fin de que una comprobación del Flag *zero* sea suficiente para establecer el final del bucle. En ciertas circunstancias la posición *0* del contador aún deberá generar una pasada. En estos casos se utiliza el Flag *sign* en vez del Flag *zero*, el cual muestra números negativos. En estos casos el bucle se interrumpe cuando se llega a *-1*.

Accesos de 16/32 bits

Siempre que sea posible se deberían utilizar accesos de 16 o hasta 32 bits a fin de limitar el número de accesos a memoria. Un punto que incrementa notablemente la velocidad es que los accesos a nivel byte de la CPU se efectúan a partir del 386 como Doubleword, por lo que dura aún más mover un solo byte que un único Doubleword.

Pero existen determinados casos en los que hay que recurrir a instrucciones de 8 bits, ya sea para copiar un número de bytes no divisible entre cuatro (9 bytes = copiar dos DWORD y después un Byte) o porque lo requiera el hardware. Las tarjetas VGA por ejemplo no permiten que se efectúen accesos más anchos de 8 bits en los modos basados en plano (como el modo x que explicaremos más adelante), ya que los registros internos de los planos (Latches) solamente tienen una anchura de 8 bits.

2.8 Instrucciones prácticas del 386

Además de las nuevas posibilidades del 386 (modo virtual, Paging, registros de 32 bits), el vocabulario general también se ha ampliado. Se han incluido algunas instrucciones muy útiles, que en parte engloban varias instrucciones del 286, con lo cual son capaces de incrementar la velocidad de forma notable en determinados puntos críticos. Utilice estas instrucciones, incluso en el modo real, y no permita que se pudran en un rincón del procesador.

Las instrucciones MOVSX y MOVZX

Vamos a explicar primero las instrucciones *MOVSX* y *MOVZX*. Ambas instrucciones son capaces de mover directamente un registro de 8 bits a un registro WORD y un registro 16 bits a un registro de 32 bits, para lo que normalmente harían falta dos instrucciones. Las letras "S" y "Z" en estas instrucciones significan "Signed" y "Zero" y se refieren a la parte superior del registro destino: Con *Movsx* se llenan en éste todos los bits o bien con *0* o con *1*, en dependencia del carácter de prefijo del registro origen, manteniéndose de esta forma los prefijos originales. La instrucción *MOVZX*, en cambio, borra la parte superior. Si *BL* tuviera por ejemplo el contenido *-1 (ffh)*, las instrucciones tendrían los siguientes resultados:

```
movzx ax,bl    ;ax contiene ahora 255 (00ffh)
movsx ax,bl    ;ax contiene ahora -1 (fffh)
```

Diferentes instrucciones SET

Bajo ciertas circunstancias las comparaciones también se pueden optimizar en un 386. Para ello este procesador posee las 30 instrucciones *SETxx*, que cubren las tareas de la combinación de *CMP*, salto condicional y *MOV*. En ellas cada salto condicionado posee un homónimo en una instrucción *Set (SETz, SETnz, SETs etc.)*. Si ahora se cumple la condición, el operador de byte traspasado se situará en 1, sino en 0.

```
dec cx          ; reducir el contador (de bucle)
set al          ; utilizar al como flag
```

En este ejemplo, que podría haber sido extraído de un bucle, *al* se sitúa normalmente en *0 (CX=0)*, y al final, cuando CX se convierte en 0, se coloca AL en 1. De esta forma también se pueden colocar directamente Variables booleanas de Pascal según una condición en ensamblador (*SETxx byte ptr Variable)*.

Multiplicaciones y divisiones rápidas con las instrucciones SHRD y SHRL

El 386 ofrece por primera vez la posibilidad de calcular directamente en registros de 32 bits (sobre todo multiplicar y dividir), lo cual es bastante más rápido que el camino convencional utilizando *DX:AX*. Pero ¿ cómo conseguir colocar números que son pasados en el formato *DX:AX* en un registro ampliado (por ejemplo, *EAX*)?

Por desgracia tampoco se puede acceder a la parte superior de estos registros, pero para ello el 386 también tiene sus instrucciones preparadas, las cuales pueden hacer mucho más que cargar registros: SHLD y SHRL , las instrucciones de desplazamiento (shift) ampliadas.

Estas instrucciones esperan -junto al número de bits a desplazar- un operando en vez de dos. Para ello se desplaza primero el primer operando (operando destino) por la cantidad correspondiente de bits; pero en vez de situar los bits liberados en el bit inferior o superior en *0*, estos se extraen del segundo operando (origen) rotado, sin modificarse este.

Si *AX* tiene por ejemplo el contenido 3 *(0000 0000 0000 0011b)* y *BX* el contenido 23 *(0000 0000 0001 0111b)*, la instrucción *SHRD BX,AX,3* rotará primero *BX* por tres hacia la derecha (=2). Al mismo tiempo se rellena el borde superior desde *AX*, de forma que el resultado del operando destino (*BX)* sea *0110 0000 0000 0010b=6002h=24578*.

La utilización más común de estas instrucciones es tal como se ha dicho la carga de registros de 32 bits (aquí *EBX)* desde dos registros de 16 bits (en el ejemplo *DX:AX)*, lo que se realiza de la siguiente forma: Primero se carga la parte superior mediante un *SHRD: SHRD EBX,EDX,16d*. Esta instrucción desplaza el *DX* "desde arriba" al registro *EBX*. A continuación se carga directamente la parte inferior con el valor deseado, manteniéndose invariable la parte superior ya colocada: *MOV BX,AX*. Este método también se aplica en el procedimiento *Wurzel* del apartado 2.3.

Multiplicación ampliada con IMUL

A ser posible también se deberían utilizar las nuevas instrucciones de multiplicación: a partir del 386 se puede multiplicar en la práctica cada registro con cualquier

valor: *IMUL DX,3* (multiplicar DX por 3) es posible de la misma forma que *IMUL AX,DX,3* (multiplicar DX por 3 y dejar el resultado en *AX*). De esta forma se puede ahorrar bastante trabajo de administración frente a las antiguas instrucciones IMUL ya conocidas.

Utilización de instrucciones 386 en programas Pascal

Todas las instrucciones 386 tienen un problema en común: Borland Pascal es incapaz de procesarlas, ni a través del ensamblador interno ni a través de programas externos conectados (si se enlaza un código de objeto mediante una directiva *$L* la indicación de procesador utilizada deberá corresponder a la utilizada en Pascal).

Así solamente queda la posibilidad de engañar al compilador mediante la inclusión de un ensamblador Inline. Para ello se actúa de la siguiente manera: se arranca el Turbo Debugger y se introduce la instrucción deseada en su forma definitiva. A continuación el Debugger mostrará al lado los códigos hexadecimales de esta instrucción. Se copian estos códigos y se incluyen en el programa después de una directiva *db*, por ejemplo:

```
db 66h,0fh,0ach,0d3h,10h ;shrd ebx,edx,16d
```

Los cambios efectuados de esta forma no son posibles así como así. O bien hay que repasar la estructura interna (aquí por ejemplo no plantea ningún problema convertir el operando *16d (10h)* en *8*, sobrescribiendo el último byte de orden con *8)*, o si no habrá que ensamblar "a mano" la orden correspondiente con el Turbo Debugger.

Lo único que nos queda es esperar que los desarrolladores de Borland se hagan a la idea de que el 386 es el estándar (o casi ya no lo es), y que por lo tanto debería ser soportado.

3. Esto arranca: ensamblador en la práctica

Este capítulo está dedicado a la aplicación práctica de la programación en ensamblador. Aquí mostraremos con ejemplos todo lo que se puede conseguir en un PC mediante la programación en ensamblador. Explicaremos la programación de los puertos serie y paralelo, del altavoz interno (también para muestreos) y de TSR.

3.1 El puerto paralelo

El puerto paralelo es seguramente para la mayoría de los usuarios, junto al teclado y la pantalla, el puerto más importante, ya que representa la conexión con la impresora. Este ámbito de aplicación es soportado de forma espléndida por el BIOS, por lo que no será necesario escribirlo de nuevo. Pero el puerto paralelo es capaz de mucho más, puede ser utilizado para el traspaso de datos a otros ordenadores y con la utilización de pocos elementos electrónicos pasivos hasta como tarjeta de sonido. Físicamente el puerto paralelo es un conector de 25 polos del tipo Sub D, con la siguiente configuración:

1	-Strobe
2-9	D0-D7
10	-Ack
11	-Busy
12	PE
13	SLCT
14	-Auto LF
15	-Error
16	-Reset
17	SLCT IN
18-25	Masa

Programación directa del puerto paralelo

Cada puerto paralelo posee tres registros situados en direcciones de puerto consecutivas. Las direcciones de puerto que se van a utilizar vienen determinadas por la llamada dirección base. Esta suele ser para el primer puerto paralelo la *378h* y para el segundo la *278h*. Estos valores pueden variar según el ordenador (una tarjeta Hercules con puerto paralelo integrado se coloca como primera conexión en el puerto *3bch*). Cuando no se esté seguro de qué puerto se va a utilizar, se podrá encontrar a partir de la dirección *0:0408h* un registro Word para LPT1, LPT2 etc., el cual indica la dirección base.

En esta dirección base se encuentra normalmente el registro de datos. En este registro se escriben los datos (es solamente de escritura) que deben ser enviados a través del puerto. Para cada *1* que se coloque aquí se sitúa el correspondiente cable en *High* y para cada *0* en *Low*.

En la siguiente dirección de puerto se encuentra el registro de estado (status), en el cual se pueden leer los mensajes de vuelta de la impresora. Para ello se transmiten las correspondientes órdenes del cable de impresora de forma directa a este registro (solamente el *busy* se representa de forma invertida en este registro). Este registro es solamente de lectura (*read only*).

Bit	Significado
7	-Busy (0= en estos momentos la impresora no puede recibir datos)
6	Ack (0= la impresora ha leído los caracteres)
5	PE, Paper empty (1= no queda papel)
4	SLCT (0= la impresora está desconectada)
3	Error (0= se ha producido un error)
2-0	reservados

El flag *Busy* indica que la impresora está ocupada (*busy*) y que no puede recibir más datos (por ejemplo cuando está lleno el búfer de la impresora). Acknowledge (*Ack*) siempre se sitúa en *0* cuando la impresora ha recibido el carácter de los cables y por lo tanto puede ser borrado. *PE* y *Error* representan errores que deben ser transmitidos al usuario para que los solucione. *SLCT* representa el estado actual del botón *En línea* de la impresora. Si está desactivado (off-line) no podrá recibir datos. Como último registro cada puerto paralelo posee un registro de control que pone a disposición vías de control en dirección a la impresora. Puede ser leído y escrito y tiene la siguiente estructura:

Bit	Significado
7-5	reservado
4	IRQ enable (1= IRQ activa)
3	SLCT (0= desconectar la impresora (off-line)
2	Reset (0= ejecutar un Reset de la impresora)
1	Auto LF (1= la impresora añade después de un CR un LF
0	Strobe (0= Hay datos esperando)

Situando el bit 4 se puede activar en teoría la *IRQ 5 o 7*, la cual se activa con cada señal de Acknowledge de la impresora. Para evitar complicaciones con la tarjeta de sonido esta capacidad suele estar desactivada. Además la conexión paralela ya se suele utilizar solamente en procedimientos de *polling* (el procesador espera a la modificación de un flag). A través de la conexión *SLCT* también se pueden situar

algunas impresoras desde el ordenador en estado off-line, y a través de la conexión Auto-LF también se puede controlar automáticamente la señal de alimentación de línea (no en todos los modelos). Todas las impresoras utilizan las conducciones *Reset y Strobe*. Esta última comunica al receptor que existe un byte en las conducciones de datos.

Acceso a la impresora

Como se ve, es muy fácil sacar un carácter por el puerto paralelo: espere a que esté situado el bit de *Busy* (conexión Busy inactiva), escriba el carácter al puerto de datos, envíe una señal *Strobe* (establecer la conexión y volver a borrarla inmediatamente) y espere la señal de Acknowledge. Como demostración encontrará un programa que efectúa esto con todos los caracteres de una cadena de ejemplo, programa situado en el CD bajo el nombre de PAR_TEST.PAS:

```
Const Basis=$378;                {dirección base del puerto paralelo}
Procedure PutChar_Par(z:Char);
{envía un carácter al puerto paralelo (dirección base en «Basis»)}
Begin
  While Port[Basis+1] and 128 = 0 Do; {esperar al final del busy}
  Port[Basis]:=Ord(z);               {colocar en el port}

  Port[Basis+2]:=Port[Basis+2] or 1;  {enviar Strobe}
  Port[Basis+2]:=Port[Basis+2] and not 1;
  While Port[Basis+1] and 64 = 1 do;  {esperar Ack}
End;

Procedure PutString_Par(s:String);
{envía cadena en el puerto paralelo, emplea PutChar_Par)}
Var i:Integer;                     {contador de caracteres}
Begin
  For i:=1 to Length(s) do         {cada carácter}
    PutChar_Par(s[i]);             {enviar al puerto paralelo}
End;

Begin
  PutString_Par('Hola, prueba de impresora MARCOMBO'#13#10);
  PutString_Par('abcdefghijklmnopqrstuvwxyz0123456789'#13#10);
End.
```

3.2 Otras aplicaciones

El puerto paralelo no tiene este nombre porque sí, en vez de puerto de impresora, ya que con él se puede ejecutar casi cualquier cosa: el puerto posee ocho salidas puras (conexiones de datos), cinco entradas puras (conexiones de control) y cuatro canales de I/O (-input/output, entrada/salida-, en el registro de control), los cuales sirven de entrada o salida según la instrucción que se utilice.

Una aplicación muy usual es la conexión de dos ordenadores a través de un cable paralelo de módem nulo con la ayuda del controlador *Interlnk.exe*, que viene incluido en el DOS desde la versión 6.0. Este controlador utiliza el protocolo de transmisión de 4 bits, el cual emplea como salida los registros de datos *D0 a D3* y como entrada las conexiones *Error, SLCT, PE y ACK*.

La función de la conexión *Strobe* ("anunciar" datos) utiliza como salida *D4* y como entrada *Busy*. El cable se conecta, por lo tanto, de la manera que se muestra a continuación:

Pin		Pin
2	con	15
3	con	13
4	con	12
5	con	10
6	con	11
15	con	2
13	con	3
12	con	4
11	con	6
10	con	5

Con este cable también se pueden utilizar otros programas comerciales de transmisión, que suelen diferenciarse solamente en el uso del software por lo que este tipo de cable es casi un estándar. Lógicamente también se pueden escribir programas de transmisión propios, aunque hoy en día el mercado ya ofrece suficiente soporte en software.

Otra posibilidad que actualmente ya no se utiliza tanto es utilizarlo como sustituto de una tarjeta de sonido, ya que en el fondo el puerto paralelo no es más que una salida digital de 8 bits. Sus datos solamente tienen que ser convertidos en señales analógicas, y ya se podrá disfrutar del sonido.

Para la conversión analógico/digital se pueden utilizar los correspondientes chips, pero entonces ya convendría comprarse una tarjeta de sonido de verdad. Resulta mucho más barato, consiguiéndose un resultado casi idéntico, utilizar unas cuantas resistencias.

Gracias a las resistencias se garantiza que el cable de mayor valor *D7* sea el primordial para la señal de salida analógica, y con la importancia decreciente del bit (D7-D0) se reduce la tensión mediante resistencias.

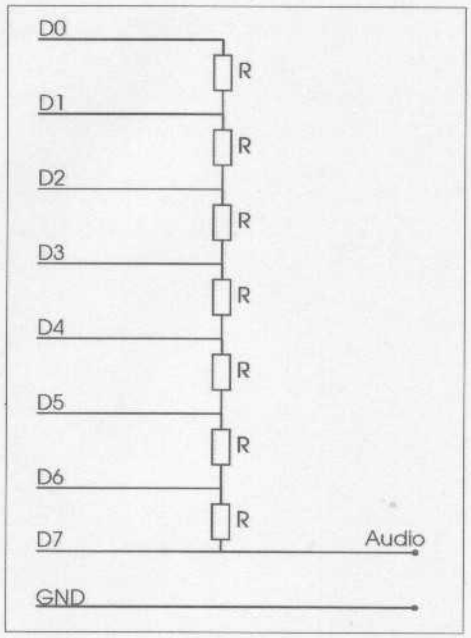

Figura 1: Conversor D/A construido con resistencias

Este adaptador (covox) se conecta directamente al puerto paralelo y a un amplificador. Ahora podrá ser utilizado por casi todos los reproductores MOD, ya que su control es increíblemente simple: los datos de sonido utilizan un determinado número de bytes por segundo (hasta 44000) que recrean el recorrido de la onda analógica. Gracias a que el PC utiliza un formato sin prefijo (a diferencia del Amiga) estos datos pueden ser transmitidos directamente al puerto paralelo (puerto de datos, en LPT1 normalmente en la dirección de puerto *378h*, véase arriba). En este caso los demás registros no hay que tenerlos en cuenta, ya que solamente sirven para la comunicación y control de impresoras y similares. Las conducciones de datos reflejan exactamente el contenido del puerto de datos.

Si se tiene en cuenta el ratio de muestreo (sampling) y se transmiten los datos a esta velocidad (p.ej. ratio de muestreo 22 kHz, 220000 Bytes por segundo al puerto), se oirá en la salida la señal audio, aunque al procesador le cueste bastante, ya que los accesos al puerto son "eternos". Por esta razón no es muy recomendable añadir operaciones gráficas al reproducir datos de audio. En estos casos es más conveniente comprarse una tarjeta de sonido.

Figura 3: Conector DA4 conectado con resistencias

4. El conocimiento de los gráficos desde el submundo

A continuación presentamos una pequeña colección de los términos técnicos más importantes de la programación de gráficos en el PC. Esta lista no pretende ser completa y solamente pretende dar un pequeño resumen a los "novatos", a fin de que se "enteren de la película". Seguramente los lectores profesionales criticarán las descripciones simplificadas, pero de lo que se trata es de dar una pequeña introducción para los principiantes, y no para los licenciados en ingeniería.

4.1 Conceptos técnicos que hay que conocer

He aquí los distintos conceptos técnicos junto con su correspondiente explicación:

Pixel, Pel (punto)

Todas las pantallas utilizadas hoy en día en el mundo de los ordenadores componen su imagen a base de píxeles. En el modo gráfico estos píxeles pueden ser manipulados individualmente en su color, mientras que en el modo texto se modifican siempre letras enteras, compuestas de varios píxeles.

Paleta

Todos los modos de 256 colores de la VGA utilizan una paleta, es decir que en vez del TrueColor (16,7 millones de colores) o la representación en HiColor (65536 colores), en las que cada punto de la imagen contiene determinados valores de los colores base rojo, verde y azul, en el modo paleta se utiliza para cada punto una especie de puntero a un registro de la paleta. Esta paleta contiene para cada uno de los 256 colores los valores correspondientes de los 3 colores base, que por lo tanto son iguales para todos los píxeles de este color. Este procedimiento tiene la ventaja de ahorrar mucha memoria, ya que en vez de los 18 bits de los registros de la paleta en la memoria de vídeo solamente se utilizan 8 bits por punto. En el modo gráfico con mucha profundidad de color esta ventaja ya no es válida, ya que la paleta se volvería gigantesca y debido a que ya se utilizan 16 o 24 bits por punto.

Otras ventajas de los modos basados en paletas se aprovechan en algunos efectos gráficos, ya que se puede asignar de una vez un color nuevo a, por ejemplo, todos los colores número 1, con sólo modificar el registro de la paleta (3 bytes). De esta forma se pueden producir transiciones de imágenes de una manera muy simple: no hace falta modificar o reducir la intensidad de cada punto, sino que basta con incrementar o reducir los 256 registros de la paleta desde 0 a su valor máximo o viceversa. Este y otros efectos basados en paletas los encontrará en el capítulo 6.

Sprite

Un sprite es en el fondo una pequeña parte de la imagen que puede situarse libremente sobre la pantalla, pero que en algunas partes puede ser transparente, con lo cual se puede mover por encima de un fondo. Algunos ordenadores personales poseen chips especiales que se encargan de esta tarea, lo que tiene la ventaja de reducir el trabajo de cálculo de la CPU, pero que por otro lado precisa unos valores determinados, p.ej. el tamaño del sprite.

Rayo catódico

Es una rayo generado por electrones acelerados, que producen en el punto del impacto sobre la pantalla un punto iluminado de un color e intensidad determinados. La imagen se crea por líneas de izquierda a derecha (visto desde delante) y de arriba hacia abajo.

Retrace (retorno de rayos)

Es el movimiento del rayo catódico durante la creación de la imagen. Se diferencia entre retrace vertical y horizontal. El retrace horizontal aparece después de la construcción de una línea y significa el salto rápido del rayo al principio de la siguiente línea. El retrace vertical se ejecuta cuando el rayo llega al borde inferior de la pantalla y vuelve a ser movido al principio de la primera línea.

En principio solamente se deberían efectuar modificaciones en el contenido de la pantalla, sean del tipo que sean (incluidos cambios en determinados registros de la VGA), durante un retrace, a fin de evitar colisiones entre modificaciones y generación de la imagen, lo que produciría distorsiones en el punto en concreto. Da igual si se espera a un retrace vertical u horizontal, aunque en el caso de grandes modificaciones o manipulaciones de registros, para las cuales la VGA tarda un poco más, se debería utilizar preferentemente el retrace vertical, ya que dura bastante más (200 veces más).

Para esperar al retrace vertical se utiliza el procedimiento *WaitRetrace* (en el módulo ModeXlib.asm, pero sirve en general para cualquier modo gráfico o de texto). Este procedimiento aprovecha que el *Input Status Register 1* (dirección de puerto *3dah)* ofrece la señal de retrace vertical en su *Bit 3*, de forma que se puede detectar en cualquier momento si se está ejecutando un retrace.

Para ello no solamente se debe esperar hasta que se señale un retrace, ya que es posible que al iniciarse un procedimiento *WaitRetrace* se esté ejecutando justo un retrace que esté a punto de finalizar. En este caso el *WaitRetrace* finalizaría inmediatamente y se efectuaría la modificación de la pantalla, para lo cual quizás ya no queda tiempo suficiente. Por esta razón el *WaitRetrace* espera al inicio de un retrace

vertical, esperando un bucle *@wait1* hasta que un eventual retrace finalice, o sea que el rayo catódico vuelva a estar sobre la pantalla, esperando el segundo bucle *@wait2* por si solo al retrace.

```
WaitRetrace proc pascal far
  mov dx,3dah          ;Input Status Register 1
@wait1:
  in al,dx             ;bit 3 a 0 cuando el haz reconstruye pantalla
  test al,08h
  jnz @wait1
@wait2:
  in al,dx             ;bit 3 a 1 cuando hay retrace
  test al,08h
  jz @wait2
  ret                  ;el haz está abajo del todo
Endp
```

Double Scan

Reducción de la resolución *y* de 400 a 200 mediante una doble representación de cada línea a fin de emular el modo de 200 líneas, el cual en el fondo no es dominado por la VGA.

La velocidad es lo más importante de la programación gráfica. Desde los inicios de los ordenadores, los programadores han tenido que luchar contra la baja velocidad de la programación gráfica. A pesar de que los procesadores y las tarjetas gráficas son cada día más rápidos, hoy en día ya no se deberían producir problemas de velocidad en gráficos CGA, cosa que si sucedía hace algunos años. Pero para mantener la calidad gráfica usual en nuestros días habrá que utilizar algunos trucos. La programación directa de los chips gráficos es inevitable, en la cual la ventaja de la velocidad frente a las rutinas del BIOS se debe sobre todo a la ausencia de comprobaciones de seguridad. Por esta razón los procedimientos de este libro no incluyen ninguna comprobación de coordenadas. Así es posible situar un punto en las coordenadas (5000/7000), lo cual puede generar los resultados más salvajes que nos podamos imaginar.

Pero dado que los programadores suelen mantener el control sobre todos los puntos individuales, una comprobación parece superflua. Si a pesar de ello fuera necesario introducir comprobaciones de seguridad, por ejemplo, cuando un usuario activo participe en el juego (con el ratón), deberá efectuarse la comprobación fuera de las rutinas de representación (*PutPixel, etc.*), a fin de no tener que aplicarla para los puntos "seguros".

4.2 La base - Modo 13h del BIOS

Al desarrollar la VGA la empresa IBM creó una buena posibilidad para acceder a la memoria de vídeo en el nuevo modo de 256 colores, la cual fue aprovechada inmediatamente por los programadores del BIOS en el modo de vídeo *13h*: el encadenamiento (chaining) de los planos de bit en un espacio de dirección lineal.

Organización de la memoria

Este espacio de memoria se inserta para la CPU (o sea también para el programador del modo *13h*) en la memoria del sistema a partir del segmento a000 y tiene la estructura más simple que se puede imaginar: a cada pixel se le asigna un byte que contiene el color, o mejor dicho el puntero a un registro de la paleta de colores. El direccionamiento sigue a la creación de la imagen por parte del rayo catódico, por lo que el punto con las coordenadas (0/0) se encuentra en el offset 0, el punto (0/1) en el offset 320 etc., hasta el punto (319/199) que se encuentra en el offset 63999. Es decir, que el direccionamiento a un punto se puede efectuar de una manera muy simple, tal como mostramos con la siguiente fórmula:

```
Offset = Y * 320 + X
```

Con esto ya se podría programar un simple desplazador de estrellas que coloca los puntos según un esquema determinado y los vuelve a borrar:

```
Uses Crt;
Var Estrellas:Array[0..500] of Record
                                x,y,Capa:Integer;
                              End;
  st_nr:Word;

Procedure PutPixel(x,y,col:word);assembler;
{coloca punto (x/y) a color col (Modo 13h)}
asm
  mov ax,0a000h                 {cargar segmento}
  mov es,ax
  mov ax,320                    {Offset = Y*320 + X}
  mul y
  add ax,x
  mov di,ax                     {cargar offset}
  mov al,byte ptr col           {cargar color}
  mov es:[di],al                {y activar punto}
End;

Begin
  Randomize;                    {inicializar números aleatorios}
  asm mov ax,13h; int 10h End;  {activar modo 13h}
```

```
Repeat                {se ejecuta una vez por construcción de pantalla}
  For St_nr:=0 to 500 do Begin {nueva posición para cada estrella}
    With Estrellas[st_nr] do Begin
      PutPixel(x,y,0);          {borrar punto antiguo}
      Dec(x,Capa shr 5 + 1);    {seguir moviendo}
      if x <= 0 Then Begin      {¿ha salido por la izquierda?}
        x:=319;                 {inicializar de nuevo}
        y:=Random(200);
        Capa:=Random(256);
      End;
      PutPixel(x,y,Capa shr 4 + 16);  {fijar nuevo punto}
    End;
  End;
  Until KeyPressed;             {funciona hasta pulsar tecla}
  TextMode(3);
End.
```

Lo importante aquí es el bucle interno que borra para cada una de las estrellas la posición anterior en la pantalla y desplaza a continuación la estrella según su velocidad (que se calcula a partir del plano). Si se sobrepasa el borde izquierdo (x<=0) se coloca en el borde derecho con nuevos valores aleatorios para la velocidad y la coordenada-y.

A continuación se sitúa el punto en su nueva posición, teniendo en cuenta su plano, lo que significa que las estrellas más lentas siempre están más atrás, o sea, que aparecen más oscuras. Para ello este programa aprovecha el hecho de que la paleta estándar, que la carga cualquier VGA cuando se enciende, contiene en los registros 16 (negro) a 31 (blanco) un degradado de grises, por lo que no es necesario crearlo.

La administración interna del modo 13h

El recorrido lineal de la memoria que facilita la programación del modo *13h* en el fondo solamente es una simulación ante la CPU. Internamente la VGA convierte la dirección lineal en una dirección plana. Para ello se utilizan las dos conducciones de dirección inferiores (bit 0 y 1 del Offset) como selección del plano de lectura o escritura, y los demás 6 bits (2-7) se utilizan, después de colocar los bits 0 y 1 a 0, como dirección física dentro del plano.

Todos los modos de texto utilizan un procedimiento similar (*Direccionamiento Odd/even (Par/Impar)*). Aquí se utiliza la conducción de dirección inferior para la selección entre el plano 0 y 1, de forma que para la CPU el byte de carácter y el byte de atributo se encuentran uno detrás de otro, pero colocándose internamente los caracteres en el plano 0 y los atributos en el plano 1, sirviendo los planos 2 y 3 como memoria del juego de caracteres.

4.3 El formato de imagen GIF

GIF es el formato gráfico más extendido en el mundo del PC, y ello tiene sus razones. Fue desarrollado en 1987 por los promotores de CompuServe para el intercambio rápido y económico de imágenes entre redes de correo electrónico y ofrece una gran cantidad de ventajas frente a otros formatos como por ejemplo PCX. Lo importante para el intercambio de datos es que el formato se definiera de forma independiente del sistema, por lo que este formato no está ligado a formatos gráficos determinados sino que contiene los datos para todos los formatos que se puedan utilizar en sistemas gráficos. El formato GIF permite una resolución de 16000 x 16000 (!) puntos de imagen con una paleta desde 256 hasta 16,7 millones de colores, y además se pueden guardar la cantidad de imágenes que se desee con una misma paleta de colores (global o local) dentro un solo fichero, cosa que por otro lado no se suele utilizar mucho.

Pero la gran ventaja es otra: GIF permite una extraordinaria compresión de las imágenes con una velocidad de compresión altísima. Se utiliza el procedimiento de compresión LZW modificado, en el cual se basan también los programas de compresión convencionales. Esta es la razón por la cual todos los programas presentados en este libro cargan las imágenes en este formato: las demos se mantienen dentro de los límites en cuanto a necesidades de memoria y a pesar de ello cargan las imágenes con bastante velocidad.

El estándar del mundillo

El formato GIF utiliza una estructura de bloque que a pesar de no estar tan definida como en el formato TIFF permite un manejo fácil de los datos marco como son la resolución y la cantidad de colores. Su estructura básica se muestra en la siguiente tabla:

Offset	Longitud	Contenido
0	3	Especificación de formato "GIF"
3	3	Número de versión, actualmente "87a"/ "89a"
6	7	Logical Screen Descriptor Block
0dh	n	Global Color Map (opcional), para modos de color de 256 colores con paleta compatible VGA (n=768 bytes)
30dh	n	Extension Block (opcional)
30dh	10	Image descriptor block
317h	n	Local Color Map (opcional), para modos de color de 256 colores con paleta compatible VGA (n=768 bytes)
?	1	Denominación de final (terminator)(89h)

En las indicaciones se partió de la base que existía un *Global Color Map*, pero faltaban los *Extension Blocks y Local Color Maps*. En vez de la *denominación de final* (terminator) se pueden añadir todos los *Image Descriptor Blocks* que se deseen, con sus datos de paletas y de tramado, indicándose con el *Terminator* el final del archivo. Dado que pocas veces se guardan varias imagenes en una archivo, y ya que aquí no se va a desarrollar un visualizador de GIF universal, sino una rápida rutina de carga para una imagen, partiremos a partir de ahora de la base de que se trata siempre de una imagen individual.

Después de la identificación de formato de 6 bytes *GIF87a o GIF89a* sigue el *Logical Screen Descriptor Block (LSDB)*, el cual define la pantalla lógica y con ello la resolución y los datos de color. La estructura de este bloque se especifica en la siguiente tabla:

Offset	Longitud	Contenido
0	2	Anchura de pantalla
2	2	Altura de pantalla
4	1	Flag de resolución:
		Bit 7: 1=Existe Global Color Map
		Bits 6-4: Profundidad de color en Bit (reducido en 1)
		Bit 3: reservado (0)
		Bits 2-0: Cantidad de Bit por pixel (reducido en 1)
5	1	Color de fondo (número de color de la paleta)
6	1	Ratio de aspecto del pixel (pixel aspect ratio)
		Bit 7: orden de la paleta global
		Bits 6-0: Pixel Aspect Ratio

A este bloque sigue la paleta global (*Global Color Map*), que contiene según el esquema VGA para cada uno de los 256 colores un registro de 3 bytes con los valores de los colores base Rojo, Verde y Azul. Hay que tener en cuenta que para cada componente de color existen 8 bits, de forma que en total se pueden seleccionar 2^{24}= 16,7 millones de colores. Al representar en una VGA deberá desplazarse cada valor en 2 bit hacia la derecha, antes de enviarlos al *VGA-DAC*, ya que la VGA solamente tiene en cuenta los 6 bits inferiores de cada componente.

Otra característica especial, que normalmente no tiene importancia en un PC, es el orden de ordenación de la paleta: las paletas globales y locales pueden estar ordenadas en el orden VGA, o sea primero el color *0*, descrito por los colores rojo, verde y azul, luego el color *1*, etc. - lo cual es la ordenación normal. Por otro lado existe la posibilidad -utilizada en raras ocasiones- de ordenar la paleta según la frecuencia del color, o sea guardar primero las partes del color rojo de los colores *0 a 255*, después todas las partes del verde y finalmente las partes de azul.

Hasta aquí todos los datos son globales, válidos para todas las imágenes que estén contenidas en este archivo GIF. Todo lo que sigue ahora es específico de una imagen y tiene que volver a ser extraído del archivo después de cada imagen leída.

Luego existen los Extension Blocks que solamente están definidos en cuanto a que están sujetos a un determinado esquema, pudiendo contener cualquier tipo de datos; algunos programas de escáner o de dibujo colocan aquí, por ejemplo, mensajes de copyright. Para un simple cargador de GIF como el que queremos crear en este capítulo los datos de este tipo le importan poco, aunque debe ser capaz de leerlos. El esquema que hemos mencionado antes consiste en que primero aparece una identificación indicando que se trata de un Extension Block. Esta identificación consta en este bloque de un signo de exclamación (*"!", ASCII 21h*). A continuación sigue un código de función no definido y la longitud del bloque de datos en sí. Dentro de un bloque de extensión se pueden encontrar una cantidad de bloques de datos variable, los cuales poseen cada uno una indicación de su longitud, señalizando un *0* el final de todo el bloque de extensión. Sigue la descripción de imagen de la imagen local en el llamado Image Descriptor Block, que tiene la siguiente estructura:

Offset	Longitud	Contenido
0	1	",", (2ch) Image Separator Header
1	2	coordenada-x de la esquina superior izquierda de la pantalla lógica.
3	2	coordenada-y de la esquina superior derecha de la pantalla lógica.
5	2	anchura de la imagen en píxeles
7	2	altura de la imagen en píxeles
9	1	Byte de Flag
		Bit 7 :1 = Existe Local Color Map
		Bit 6 :1 = Imagen está entrelazada
		Bit 5 : Orden de la paleta local
		Bits 3-4 : reservado (0)
		Bits 0-2 : Bit por Pixel (reducido en uno)

Lo que es interesante aquí es el *FlagByte*. En este punto se indica si al bloque descriptor de imagen (Image Descriptor Block) sigue una paleta local, que tiene preferencia sobre la paleta global, y también se pueden indicar de forma especial el tipo de ordenación y la cantidad de píxeles. El *Bit 6* indica si la imagen está entrelazada, lo que tiene el mismo efecto que la característica del mismo nombre de los monitores: primero se describen todas las líneas de imagen pares (0,2,4) y a continuación las impares. Esta capacidad se ideó para conseguir una imagen aproximada durante la bajada o transmisión de datos desde soportes lentos como BBS o CompuServe, o también desde un disquete.

A este bloque le sigue inmediatamente, siempre que exista, la paleta local, la cual contiene las mismas indicaciones que la paleta global. Después vienen los Raster Data Blocks, que son los que contienen en formato LZW los datos de la imagen en sí. Los datos se guardan, pues, de la misma forma que la descripción de la imagen en formato de bloques, utilizándose para el byte de longitud solamente *8 bits*, lo cual limita el tamaño máximo de un bloque a 256 bytes (incluyendo el byte de longitud).

El sistema de compresión LZW

El primer bloque raster contiene una característica especial: aquí se indica directamente antes del byte de longitud cuántos bit describirán un píxel. Esto tiene que ver con el sistema de compresión LZW, el cual construye sus algoritmos en base a este tamaño.

La mayor ventaja para nuestras intenciones es, sin lugar a dudas, el reducido tamaño de archivo de imagen que se consigue con la compresión LZW (desarrollada por Lempel, Ziv y Welch). Este procedimiento no es el sistema de compresión más rápido, pero en cambio puede ser descomprimido con mucha velocidad.

Frente a los simples sistemas de compresión como RLE (Run Length Encoding), que se utiliza en el formato PCX, el LZW no sólo es capaz de juntar bytes similares que van seguidos, sino cualquier cadena de bytes aunque no vayan seguidas. Esto se consigue gracias a la utilización de un alfabeto extendido. Este alfabeto utiliza, en vez de los 8 bits que se usan normalmente, más bits para la codificación. Si se utilizan por ejemplo 9 bits por carácter, se podrán colocar en el archivo además de los códigos normales *0-255* también los códigos *256-511*, que luego se cargan con cadenas de caracteres completas. Esta parte ampliada del alfabeto no se ocupa de entrada, sino que se genera durante la compresión (o descompresión). El algoritmo de compresión es en el fondo bastante simple:

Va leyendo caracteres del archivo origen (sin comprimir) (o de la memoria de vídeo), hasta que la cadena de caracteres leída ya no esté en el alfabeto. Esto se produce al principio, después de dos caracteres, ya que el primer carácter aún se encuentra en el alfabeto (solamente es un carácter con el contenido *0 a 255)*, pero la cadena compuesta por los dos primeros caracteres aún no existe.

En cuanto el compresor ha leído hasta aquí, escribe esta cadena de caracteres en su alfabeto, de forma que en su siguiente aparición la podrá sustituir por su valor alfabético, o sea, comprimirla. A continuación se escribe el valor del alfabeto de la cadena de caracteres más larga que aún existe en el alfabeto en el archivo destino y se inicializa la cadena de caracteres con el último carácter leído, al cual se vuelven a añadir tantos caracteres hasta que la cadena no exista en el alfabeto.

Ahora se plantea la pregunta de cuántos bits se deberían utilizar para el alfabeto. Si se eligen demasiado pocos, se desborda rápidamente y tiene que volver a ser inicializado, lo que influye negativamente en al ratio de compresión, pero si se cogen demasiados, se desperdicia sobre todo al principio bastante espacio, ya que los bits superiores no se necesitan. La solución es el procedimiento LZW modificado, el cual utiliza una anchura de bytes variable: se comienza con 9 bits, lo que permite un alfabeto con 512 registros. Si se sobrepasa este límite, simplemente se añade otro bit.

No tiene mucho sentido ir ampliando cada vez más, ya que se ocuparían demasiados bits innecesarios, pese a que la mayor parte de las cadenas contenidas en el alfabeto ya no se vayan a utilizar. Por esta razón cuando se supera una anchura de 12 bits se envía antes o después un código de borrado (clear), que borra completamente el alfabeto y restablece la anchura a 9 bits, de forma que la compresión vuelve a comenzar desde el principio.

Lo que está claro es que la eficacia del algoritmo depende bastante del tamaño del archivo que se desea comprimir. En cuanto existen grandes cantidades de datos se puede acceder más a menudo al alfabeto y se pueden colocar largas cadenas de caracteres en pocos bits. Este algoritmo es ideal para imágenes, ya que estas suelen ocupar varios Kbytes.

Para nosotros, sin duda alguna, es más importante el algoritmo de descompresión, mediante el cual se generan imágenes reconocibles a partir de los datos comprimidos; pero para entender el procedimiento de compresión hay que explicar cómo funciona. Lo interesante del procedimiento LZW es que no hace falta incluir en la grabación el alfabeto, ya que se recupera de los datos empaquetados durante la descompresión. Aquí se aprovecha que en los datos empaquetados solamente aparecen cadenas de caracteres codificadas que ya han aparecido con anterioridad y por lo tanto están presentes en el alfabeto.

El descompresor actúa de la siguiente forma: Cada carácter (comprimido) leído se comprueba primero a ver si se trata de un byte concreto sin comprimir, lo que se descubre si su valor es inferior a 256. Estos caracteres pueden ser escritos directamente al archivo destino (o la memoria de vídeo). Si se trata en cambio de un código extendido se recupera la cadena de caracteres correspondiente del alfabeto y se escribe. Para ello lógicamente tendrá que montarse al mismo tiempo el alfabeto, combinando la ultima cadena de caracteres descomprimida (o el carácter sin comprimir) con el primer carácter de la cadena de caracteres que se está descodificando en ese momento.

Esto es exactamente el mismo procedimiento que el del compresor, solamente "al revés". De esta forma los alfabetos se corresponden en cada momento durante la compresión y descompresión; en casi cada momento, ya que existe (como siem-

pre) una excepción, en la cual aparece un carácter cuyo código aún no está incluido en el alfabeto: si se comprime una cadena de caracteres con la forma *AbcAbcA*, y la cadena de caracteres *Abc* ya existe en el alfabeto, el compresor escribirá el código de este registro en el archivo destino y generará el nuevo registro del alfabeto *AbcA*, el cual vuelve a aparecer inmediatamente después, utilizándose y, por lo tanto, escribiéndolo al archivo destino.

Pero en este momento el descompresor aún no conoce esta cadena de caracteres; cómo va a saber que el siguiente carácter que va a aparecer es una *A*, ya que no se escribe al archivo destino. Este caso solamente se produce en cadenas de caracteres como la que hemos descrito, de forma que se puede evitar de una forma muy fácil: si aparece un código que aún no está en el alfabeto, se escribe simplemente la última cadena decodificada más el primer carácter a la memoria de vídeo, y se registra la nueva cadena en el alfabeto.

En otros tiempos el problema solía ser la memoria, y justo un alfabeto puede necesitar, en determinadas circunstancias, bastante. Por esta razón el algoritmo se volvió a mejorar, lo cual parecía hacerlo más complicado, pero en realidad lo simplificaba. Tal como hemos visto cada nuevo registro del alfabeto se genera tanto en la compresión como en la descompresión a partir de una cadena de caracteres ya existente más un carácter nuevo, guardándose simplemente el código de la cadena antigua y el código del nuevo carácter, necesitándose por lo tanto solamente dos registro (llamados Prefix y Tail, Prefijo y Cola).

Cargador GIF optimizado a 320 x 200

Por lo que hemos visto, GIF es un formato universal que aglutina una gran cantidad de resoluciones diferentes junto a diversas profundidades de color, y que además es independiente del sistema. Si se desea programar un buen visualizador GIF, este deberá conocer y soportar todas las variantes del formato, algo que no ayuda mucho en cuanto a mejorar la velocidad. Pero como ya existen bastantes visualizadores GIF en el mercado shareware, el sentido de este capítulo no va a ser añadir uno más. Lo que se pretende es desarrollar un cargador optimizado para determinados formatos, que pueda compensar la falta de versatilidad con una alta velocidad.

El formato preferido hoy en día en el mundo de las demos y los juegos es sin lugar a dudas el formato 320 X 200 con 256 colores, por lo que nos vamos a poner como meta este modo gráfico, aunque, tal como veremos más adelante, no será ningún problema ampliarlo a otros modos de 256 colores. Lo que se excluye es la utilización de 16 colores, ya que el descompresor debería ser modificado completamente para ello (profundidad de color de 4 bits en vez de 8 bits). La rutina *LoadGIF* de la *Unit GIF*, que también es utilizada por muchos otros programas de demostración de este libro, nos servirá aquí como ejemplo:

```
unit gif;                          {cabecera para gif.asm}
Interface
uses modexlib;                     {SetPal}
  var
  vram_pos,                        {posición actual en la RAM VGA}
  rest, errornr:word;              {restantes bytes en memoria y error}

  gifname:String;                  {nombre, ampliado en #0}
  Procedure LoadGif(GName:String);
                                   {carga "GName.gif" en vscreen}
  Procedure LoadGif_Pos(GName:String;Posit:Word);
                                   {carga GIF en offset Posit}
Implementation
  Procedure ReadGif;external;      {cargador GIF, completamente en Asm}
  {$l gif}

  Procedure LoadGif;
  {carga archivo "GName.gif" en vscreen}
  Begin
    If pos('.',gname) = 0 then     {añadir extensión "gif" si procede}
      gname:=gname+'.gif';
    Gifname:=GName+#0;;            {crear cadena ASCIIZ}
    vram_pos:=0;                   {comenzar en la RAM VGA con offset 0}
    ReadGif;                       {y cargar imagen}
    If Errornr <> 0 Then           {cancelar en caso de error}
      Halt(Errornr);
    SetPal;                        {activar paleta ajustada}
  End;

  Procedure LoadGif_pos;
  {cargar archivo GIF en offset de pantalla Posit}
  Begin
    If pos('.',gname) = 0 then     {añadir extensión "gif" si procede}
      gname:=gname+'.gif';
    Gifname:=GName+#0;             {crear cadena ASCIIZ}
    vram_pos:=posit;               {comenzar en la RAM VGA con offset 0}
    ReadGif;                       {y cargar imagen}
    If Errornr <> 0 Then           {cancelar en caso de error}
      Halt(Errornr);
    SetPal;                        {activar paleta ajustada}
  End;
Begin
  errornr:=0;                      {normalmente sin error}
  GetMem(VScreen,64000);           {direccionar pantalla virtual}
End.
```

Los únicos procedimientos de esta unidad *LoadGIF y LoadGIF_Pos* solamente representan el marco para la llamada a la parte de ensamblador, y por lo tanto para la simplificación del proceso de carga. *LoadGIF* es el procedimiento que se utiliza normalmente para cargar una imagen en la pantalla virtual (*vscreen)* y que descarga un eventual desbordamiento (en imágenes superiores a *320 x 200)* a la memoria de vídeo a partir del *Offset 0*. Al procedimiento *LoadGIF_Pos* se le puede indicar además el offset a partir del cual comienza la descarga.

En caso necesario ambos procedimientos añaden la terminación *.GIF* al nombre de archivo traspasado y amplían la cadena con un 0 final, el cual es necesario para el DOS. Se ha añadido también un simple (!) tratamiento de errores, el cual en esta forma simplemente evita una caída del sistema finalizando el programa cuando *ReadGIF* devuelve un número de error distinto de *0*. Frente a un cargador GIF universal, en una demo se supone que sus archivos están situados en el directorio actual. Al final se activa un *SetPal* que iguala la paleta de imagen cargada. El procedimiento de carga en sí, *ReadGIF*, se encuentra en el archivo GIF.ASM, el cual consigue, gracias al puro código en ensamblador, la máxima velocidad.

```
.286

clr=256                         ;código para "borrar alfabeto"
eof=257                         ;código para "fin de archivo"
w equ word ptr
b equ byte ptr

data segment public
  extrn gifname:dataptr         ;nombre archivo, incl ".gif" + db 0
  extrn vscreen:dword           ;puntero a zona de memoria de destino
  extrn palette:dataptr         ;paleta de destino
  extrn vram_pos:word           ;posición en memoria de pantalla
  extrn rest:word               ;resto que se ha de copiar
  extrn errornr:word;           ;bandera de error

  handle      dw 0              ;Handle-DOS para archivo GIF
  Buf         db 768 dup (0)    ;búfer de los datos leidos
  BufInd      dw 0              ;puntero en el búfer
  abStack     db 1281 dup (0)   ;pila para desencriptar un byte
  ab_prfx     dw 4096 dup (0)   ;alfabeto, parte prefijo
  ab_tail     dw 4096 dup (0)   ;alfabeto, parte sufijo
  free        dw 0              ;siguiente posición libro en alfabeto
  anchura     dw 0              ;nº bits de un byte
  max         dw 0              ;long. máx. alfabeto con anchura act.
  stackp      dw 0              ;puntero en la pila del alfabeto
  restbits    dw 0              ;nº de bits a leer
  restbyte    dw 0              ;nº de bytes en el búfer
  caso_especial dw 0            ;mem. intermedia para caso especial
```

```
  Act_code   dw 0          ;código en proceso
  old_code   dw 0          ;código anterior
  readbyt    dw 0          ;byte en proceso
  lbyte      dw 0          ;último byte físico leído
data ends

extrn p13_2_modex:far      ;se necesita en caso de overflow

code segment public
assume cs:code,ds:data

public readgif
GifRead proc pascal n:word
;lee n bytes físicos del archivo
  mov ax,03f00h            ;función 3fh de Interrupción 21h: leer
  mov bx,handle            ;cargar Handle
  mov cx,n                 ;cargar nº de bytes a leer
  lea dx,puf               ;puntero a búfer destino
  int 21h                  ;ejecutar interrupción
  ret
gifread endp

GifOpen proc pascal
;abre el archivo GIF para acceso de lectura
  mov ax,03d00h            ;función 3dh: abrir
  lea dx,gifname + 1       ;puntero a nombre (saltar byte de long.)
  int 21h                  ;ejecutar
  mov handle,ax            ;guardar Handle
  ret
gifopen endp

GifClose proc pascal
;cierra archivo GIF
  mov ax,03e00h            ;función 3eh: cerrar
  mov bx,handle            ;cargar Handle
  int 21h                  ;ejecutar
  ret
gifclose endp

GifSeek proc pascal Ofs:dword
;posicionado en el archivo
  mov ax,04200h            ;función 42h,
  mov bx,w handle          ;subfunción 0: Seek rel. inicio archivo
  mov cx,word ptr Ofs + 2  ;cargar offset
  mov dx,word ptr Ofs
  int 21h                  ;ejecutar
  ret
Endp
ShiftPal proc pascal
```

```
;adapta el formato de paleta de 24 bits al formato VGA de 18 bits
  mov ax,ds                ;array fuente y destino en segm. datos
  mov es,ax
  mov si,offset Buf        ;leer del búfer de datos
  lea di,palette           ;escribir en paleta
  mov cx,768d              ;copiar 786 bytes
@l1:
  lodsb                    ;obtener byte
  shr al,2                 ;convertir
  stosb                    ;y escribir
  loop @l1
  ret
Endp
FillBuf proc pascal
;lee un bloque del archivo en Buf
  call gifread pascal,1    ;leer un byte
  mov al,b puf[0]          ;cargar longitud en al
  xor ah,ah
  mov w restbyte,ax        ;y guardar en RestByte
  call gifread pascal, ax  ;leer bytes
  ret
Endp

GetPhysByte proc pascal
;obtiene un byte físico del búfer
  push bx                  ;bx se necesita por el invocador
  cmp w restbyte,0         ;¿no quedan datos en búfer?
  ja @restda
  pusha                    ;llenar búfer de nuevo
  call FillBuf
  popa
  mov w IndBuf,0           ;y punto de vuelta
@restda:                   ;datos en el búfer
  mov bx,w BufInd          ;cargar puntero de búfer
  mov al,b Buf[bx]         ;obtener byte
  inc w IndBuf             ;puntero sigue
  pop bx                   ;y listo
  ret
Endp

GetLogByte proc pascal
;obtiene un byte lógico del búfer, emplea GetPhysByte
  push si                  ;si se necesita por el invocador
  mov ax,w anchura         ;obtener anchura de byte
  mov si,ax                ;y guardar
  mov dx,w restbits        ;desplazar lbyte 8-Restbits a la derecha
  mov cx,8
  sub cx,dx                ;formar resta
  mov ax,w lByte
```

```
  shr ax,cl                 ;y desplazar
  mov w Act_code,ax         ;guardar código
  sub si,dx                 ;Restbits ya obtenido -> restar
@nextbyte:
  call getphysbyte          ;obtener nuevo byte
  xor ah,ah
  mov w lByte,ax            ;guardar en lByte para sig. byte lógico
  dec w restbyte            ;marcar byte como recuperado

  mov bx,1                  ;enmascarar demás bits en byte obtenido
  mov cx,si                 ;fijar nº de bits
  shl bx,cl                 ;desplazar
  dec bx                    ;y decrementar
  and ax,bx                 ;enmascarar byte

  mov cx,dx                 ;desplazar a la posición correcta
  shl ax,cl                 ;es decir, Restbits a la izquierda
  add w Act_code,ax         ;y sumar al resultado

  sbb dx,w anchura          ;reducir restbits
  add dx,8                  ;en lo que pase de 8 bits
  jns @positiv
  add dx,8
@positiv:
  sub si,8                  ;obtenidos hasta 8 bits -> restar
  jle @listo                ;<= 0 -> todo listo, fin
  add dx,w anchura          ;incrementar Restbits en bits que faltan
  sub dx,8
  jmp @nextbyte             ;y seguir
@listo:
  mov w restbits,dx         ;guardar Restbits para siguiente llamada
  mov ax,w Act_code         ;y cargar ax
  pop si
  ret
Endp

ReadGif proc pascal
;carga una imagen gif de nombre gifname en vscreen, y la deposita en
;la pantalla
  push ds                   ;guardar ds
  call GifOpen              ;abrir archivo
  jnc ok                    ;¿error?
  mov errornr,1             ;notificar y terminar
  pop ds
  ret

ok:
  call gifseek pascal, 0,13d ;saltar los primeros 13 bytes
  push 768d                 ;cargar 786 bytes de la paleta
```

```
  call gifread
  call shiftpal              ;y convertir a "Paleta"
  call gifread pascal,1      ;saltar un byte

@extloop:                    ;leer Extension-Blocks
  cmp w puf[0],21h           ;¿queda algún Extension-Block?
  jne @noext                 ;no, seguir
    call gifread pascal,2    ;leer primeros dos bytes
    mov al,b puf[1]          ;longitud del bloque de datos
    inc al                   ;aumentar en 1
    xor ah,ah
    call gifread pascal, ax ;y leer
  jmp @extloop

@noext:
  call gifread pascal, 10d   ;leer resto del IDB
  test b puf[8],128          ;¿paleta local?
  je @nolok                  ;no, seguir
    push 768                 ;si no leer
    call gifread
    call shiftpal            ;y fijar

@nolok:
  les di,dword ptr vscreen   ;cargar direcc. destino

  mov w lbyte,0              ;último byte leido 0
  mov w free,258             ;primera entrada libre 258
  mov w anchura,9            ;anchura de byte 9 bits
  mov w max,511              ;con ello entrada máxima en 511
  mov w stackp,0             ;puntero de pila a inicio
  mov w restbits,0           ;no hay Restbits
  mov w restbyte,0           ;u obtener Restbits
@mainloop:                   ;recorrer para cada byte lógico
  call getlogByte            ;obtener byte lógico
  cmp ax,eof                 ;marca End of File
  jne @no_cancelar
  jmp @cancelar              ;si, fin
@no_cancelar:
  cmp ax,clr                 ;¿Clr-Code?
  jne @no_clear
  jmp @clear                 ;si, borrar alfabeto
@no_clear:
  mov w readbyt,ax           ;guardar byte actual
  cmp ax,w free              ;código ya está en el alfabeto(free)
  jb @code_in_ab             ;sí, visualizar
  mov ax,w old_code          ;no, caso especial, último string
  mov w Act_code,ax          ;a procesamiento
  mov bx,w stackp
  mov cx,w caso_especial     ;añadir ler carácter (siempre concreto)
```

```
  mov w abstack[bx],cx          ;guardarlo en la pila
  inc w stackp                  ;Stack-Pointer sigue
@code_in_ab:                    ;Code existe en el alfabeto:
  cmp ax,clr                    ;< Clr-Code ?
  jb @concreto                  ;entonces carácter concreto
@fillstack_loop:                ;si no decodificar
  mov bx,w Act_code             ;código actual como puntero en alfabeto
  shl bx,1                      ;Word-Array (!)
  push bx
  mov ax,w ab_tail[bx]          ;obtener Tail es concreto
  mov bx,w stackp               ;colocar en la pila
  shl bx,1                      ;también Word-Array
  mov w abstack[bx],ax          ;guardar
  inc w stackp
  pop bx
  mov ax,w ab_prfx[bx]          ;obtener prefijo
  mov w Act_code,ax             ;enviar a decodificar como código actual
  cmp ax,clr                    ;> Clr-Code
  ja @fillstack_loop            ;seguir decodificando
@concreto:                      ;sólo quedan valores concretos la pila
  mov bx,w stackp               ;colocar último código en la pila
  shl bx,1                      ;Word-Array
  mov w abstack[bx],ax
  mov w caso_especial,ax        ;guardar también para el caso especial
  inc w stackp                  ;puntero sigue
  mov bx,w stackp               ;preparar lectura de la pila
  dec bx                        ;decrementar puntero y
  shl bx,1                      ;alinear a Word-Array
@readstack_loop:                ;procesar pila
  mov ax,w abstack[bx]          ;obtener caracteres de la pila
  stosb                         ;y escribir en memoria destino

  cmp di,0                      ;¿Rebase de segmento?
  jne @noovl1
  call p13_2_modex pascal,vram_pos,16384d
  add vram_pos,16384d           ;colocar parte en memoria de pantalla
  les di,dword ptr vscreen      ;Posición en VGA-Ram sigue y nuevo pun-
tero destino

@noovl1:
  dec bx                        ;Stack-Pointer al siguiente elemento
  dec bx
  jns @readstack_loop           ;¿procesado? no, seguir
  mov w stackp,0                ;Stackpointer-Variable a 0
  mov bx,w free                 ;colocar en el alfabeto
  shl bx,1                      ;posicionar posición "free"
  mov ax,w old_code             ;escribir últmio código en prefijo
  mov w ab_prfx[bx],ax
  mov ax,w Act_code             ;código actual al Tail
```

```
    mov w ab_tail[bx],ax
    mov ax,w readbyt              ;guardar byte leido cómo último código
    mov w old_code,ax
    inc w free                    ;a la siguiente posición en el alfabeto
    mov ax,w free
    cmp ax,w max                  ;¿máximo alcanzado?
    ja @no_mainloop
    jmp @mainloop                 ;no, simplemente seguir
@no_mainloop:
    cmp b anchura,12              ;¿anchura ya 12 bits?
    jb @no_mainloop2
    jmp @mainloop                 ;sí, seguir
@no_mainloop2:
    inc w anchura                 ;si no incrementar
    mov cl,b anchura              ;calcular nuevo valor máximo
    mov ax,1                      ;desplazar 1 a la izq. en nueva anchura
    shl ax,cl
    dec ax                        ;y decrementar
    mov w max,ax                  ;guardar
    jmp @mainloop                 ;y de vuelta a l bucle principal
@clear:                           ;reponer alfabeto
    mov w anchura,9               ;anchura de nuevo a valor original
    mov w max,511                 ;máximo de nuevo en 511
    mov w free,258                ;primero posición libre en 258
    call getlogbyte               ;obtener siguiente byte
    mov w caso_especial,ax        ;guardar como caso especial
    mov w old_code,ax             ;y también como última leída
    stosb                         ;valor directo a memoria, concreto

    cmp di,0                      ;¿rebase de segmento?
    jne @noovl2
    call p13_2_modex pascal,vram_pos,16384d
    add vram_pos,16384d           ;guardar en memoria de pantalla
    les di,dword ptr vscreen      ;punt. RamVGA sigue y dir. inicial nueva

@noovl2:
    jmp @mainloop                 ;volver al bucle prinpipal
@cancelar:                        ;interrupción mediante código EOF
    call gifclose                 ;cerrar archivo
    mov rest,di                   ;guardar nº de bytes a copiar y
    pop ds                        ;terminar
    ret
Endp

code ends
end
```

Los primeros procedimientos de este módulo *GIFOpen, GIFRead, GIFSeek y GIFClose* sirven para el manejo del archivo, abriendo el archivo , leyendo datos (*n bytes*),

situando el puntero del archivo en la posición *Ofs* y volviendo a cerrar el archivo. Un procedimiento que no tiene que ver directamente con la carga es *ShiftPal*. Aquí se lee la paleta del búfer de datos al que fue cargada anteriormente con *GIFRead*, y se traslada al array "Palette", de forma que puede ser colocada directamente. Aquí se ejecuta también la adaptación de formato, la cual genera mediante desplazamientos (shifts) a la derecha de los valores de 8 bits del formato GIF los valores de 6 bits del formato VGA.

El siguiente nivel superior del cargador GIF consiste en el procedimiento *GetPhysByte*, la tarea del cual es extraer exactamente un byte del archivo. Esto se podría conseguir también directamente con *GIFRead*, pero los accesos individuales al disco suelen ser más lentos, pese al caché, que el almacenamiento intermedio en búferes a través del programa. *GetPhysByte* comprueba primero si aún existen datos en este búfer, cuyo grado de ocupación se muestra mediante *RestByte*. En caso afirmativo se lee simplemente el siguiente byte, en caso contrario deberá llenarse el búfer con nuevos datos mediante *FillBuf*. Este procedimiento lee la longitud de bloque actual y a continuación el número correspondiente de bytes desde el archivo.

A causa del método de compresión se origina un problema que obliga a evaluar los datos extraídos hasta el momento: un byte ya no tiene una anchura de 8 bits, sino que contiene, según el estado actual del alfabeto, por lo menos 9 bits. Por esta razón habrá que filtrar de este flujo continuado de bits (servido por *GetPhysByte* en bloques de 8 bits, o sea bytes físicos) el número de bits actual. Esta tarea es realizada por *GetLogByte*.

Dado que aquí se trabaja con bytes partidos, las variables *Lbyte* y *Restbits* juegan un papel importante: *Lbyte* contiene el último byte físico que fue servido por *GetPhysByte*. Este byte aún hay que convertirlo en bits *Restbits*. Para ello se desplaza el *Lbyte* en *8 Restbits* a la derecha y se guarda este valor en *act_Code*.

El siguiente paso consiste en recuperar otro byte, cuyos bits necesarios (inferiores) tienen que ser desenmascarados y sumados. En anchuras de byte de 10 bits puede ser que haga falta otro byte físico. En este caso se salta hacia atrás a la etiqueta *@nextbyte* y se evalúan estos bytes de la forma correspondiente.

El procedimiento más importante (y único declarado *publico*) de este módulo es *ReadGIF*. A este procedimiento se le traspasan globalmente: *GIFName*, que contiene el nombre completo del archivo con un *0* de cierre, y *vscreen*, que contiene un puntero a la pantalla virtual, que ya debe estar asignado (se ejecuta en el arranque de la unidad).

Después de abrir el archivo (con verificación de error) se saltan los primeros 13 bytes, ya que los datos que contienen no tienen mayor importancia (identificación

y Logical Screen Descriptor), ya que se presupone el formato 320 x 200. Otros formatos no se aceptan por razones de velocidad y solamente generan basura en la pantalla, pero de lo que se trata es de conseguir velocidad y no universalidad.

También se da por sobreentendido que existe una paleta global, que seguidamente se carga y se iguala mediante *ShiftPal*. Ahora se lee la identificación del siguiente bloque y un eventual bloque de expansión se ignora. Si no existe ningún bloque de expansión, se tratará del *Image Descriptor Block*. Por ello se leen ahora 10 bytes: 9 bytes que provienen del *IDB* y el primer byte del *Raster Data Block*. Si existe una paleta local (bit 7 del flag de registro IDB) se carga esta y se coloca.

Con la etiqueta *@nolok* el procedimiento entra en la descodificación en sí de los datos. Primero se inicializan unas cuantas variables: *free* contiene la primera posición libre en el alfabeto, *anchura* la anchura actual de un byte, y *max* la posición más alta que existe actualmente en el alfabeto. A continuación comienza el bucle principal (*@mainloop*) que es ejecutado para cada byte:

Cada byte leído es comparado primero con los códigos *eof* (end of file) *y clr* (Clear Alphabet). En el primer caso el archivo está cargado completamente, en el segundo se ha producido en este punto, durante la compresión, un desbordamiento definitivo del alfabeto, lo cual precisa un borrado y una nueva inicialización del mismo. Para que el compresor y descompresor trabajen de forma sincronizada, en la descompresión también deberá restablecerse el alfabeto en este punto.

A continuación se comprueba si el código leído ya existe en el alfabeto. En caso de que no fuera así, se tratará del caso especial descrito antes, que será tratado ahora de la forma correspondiente: la última cadena (existente comprimida en *Old_Code*) se combina con el penúltimo carácter guardado y se traspasa al Stack (*abstack*). Este stack se evalúa después en orden inverso al orden de su creación y se escribe en la memoria principal.

Después del tratamiento de este caso especial, y también si no se produjo ningún caso especial, se continúa en *code_in_ab*. Aquí se comprueba primero si el valor está por debajo de los códigos *Clr*, o sea dentro del alfabeto "normal" (0...255). En caso afirmativo se trata de un código concreto, el cual es desplazado a partir de la etiqueta *@concreto* al stack. En caso contrario deberá ser descodificado.

Aquí es donde entra en acción el stack nombrado. El descompresor utiliza el alfabeto comprimido, que tiene un tipo de estructura recursiva: el sufijo (*Tail*) es concreto en cualquier caso, ya que se añadió durante la compresión un carácter concreto. El prefijo, en cambio, suele estar comprimido y tiene que ser descomprimido mediante un registro del alfabeto, el cual se compone nuevamente de una parte comprimida y una sin comprimir. Este juego continúa hasta que un registro del

alfabeto tiene dos partes concretas. Hasta este punto todos los sufijos (*Tail)* han sido desplazados al stack, ya que aún no se han tenido que utilizar, y pueden ser escritos al destino.

El bucle para el desarrollo de este código está marcado por la etiqueta *@fillstack_loop*. Aquí es donde se aplica el procedimiento descrito anteriormente, desplazar el sufijo del código actual al stack y descomprimir el prefijo, siempre que esté comprimido, hasta que también sea concreto y se pueda abandonar el bucle.

En la etiqueta *@concreto* se desplaza el último prefijo al stack, se guarda éste para el tratamiento de un eventual caso especial que pudiera seguir y finalmente se vacía el stack en un bucle (*@readstack_loop)*.

En este punto (después del *stosb)* también se tratan imágenes que tengan un tamaño superior a 320 x 200. El único problema que tienen estas imágenes es su tamaño. No caben en el segmento de la pantalla virtual, por lo que se descarga a la memoria de vídeo y se desplaza el puntero dentro de ésta para eventualmente poder seguir descargando y poder escribir al final el resto a la posición correcta.

A partir de la etiqueta *@noovl1* la rutina se vuelve a dedicar a su tarea de descompresión: el bucle se repite hasta que todo el stack está vacío. A continuación aún hay que actualizar el alfabeto, registrando el último valor como prefijo y el código actual como sufijo (Tail). Ahora se actualiza el puntero del alfabeto (*free)*. En el caso de que sobrepase el límite del alfabeto, éste se amplía, se calcula el nuevo valor *max* y se continúa trabajando de forma normal. Si en cambio la anchura ya es de 12 bits, ya no se amplía más, sino que se salta de vuelta al bucle principal. Poco después el compresor enviará el código *Clr*, el cual es tratado a partir de la etiqueta *@Clear*:

Las variables que hacen referencia al alfabeto son restablecidas a sus valores originales y el siguiente carácter, el cual está descomprimido de forma garantizada, se escribe directamente. Aquí también hay que atrapar un eventual tamaño demasiado grande mediante una descarga. Finalmente se cierra el archivo y se guarda el "estado de relleno" de la pantalla virtual. El programa inicial no tendrá que hacer nada más que copiar el resto de la pantalla virtual a la pantalla real.

```
p13_2_Modex (vram_pos, rest div 4);
```

En el caso especial 320 x 200 también se pueden utilizar constantes simplificadas:

```
p13_2_Modex (0, 16000);
```

PCX - simple y rápido

Además de GIF existen muchos otros formatos con diferentes capacidades y métodos de compresión, entre ellos el PCX, que fue desarrollado por la empresa Zsoft para el programa de dibujo Paintbrush, y que en la actualidad es bastante popular. Este formato no convence ni por su posibilidad de conversión entre diferentes sistemas de ordenador ni por sus altos ratios de compresión. Lo que cuenta en el formato PCX es su simplicidad. Las datos marco son muy simples, y el algoritmo de compresión (*RLE*) tampoco es nada del otro mundo.

PCX ha sufrido diversas modificaciones desde su creación, las cuales tenían capacidades muy diferenciadas, pero a nosotros solamente nos interesa la versión 3.0, la cual soporta imágenes de 256 colores. Independientemente de su versión un archivo PCX tiene siempre una cabecera (header) de 128 bytes, que contiene información sobre el tamaño y la posición de la imagen. En la versión 3.0 esta cabecera utiliza 70 bytes, no teniendo ninguna importancia el resto, que suele estar rellenado con ceros. La estructura de la cabecera es muy simple:

Offset	Longitud	Contenido
0	1	Identificación del formato 0ah
1	1	Versión (5 para la versión 3.0)
2	1	Utilizada compresión RLE (1) o no (0)
3	1	Bits por Pixel
4	4	Coordenadas de la esquina superior izquierda (x/y como words)
8	4	Coordenadas de la esquina inferior derecha (x/y como words)
12	2	resolución horizontal en dpi (normalmente la anchura de la imagen)
14	2	resolución vertical en dpi (normalmente la altura de la imagen)
16	48	Paleta (16 colores, no suele tener importancia con 256 colores)
64	1	reservado
65	1	número de planos (con 256 colores un plano)
66	2	Bytes por línea de imagen, redondeado a valor entero.
68	2	Tipo de paleta (1 = color o b/N, 2 = escala de grises)
70	58	reservado (casi siempre 0)

Después de la identificación de formato y el número de versión sigue un flag para la compresión: *0* significa que no se ha utilizado compresión, lo cual puede producir en ciertas circunstancias archivos más pequeños, que ya explicaremos más adelante. Un *1* significa que se ha utilizado una compresión *RLE*.

Después de la profundidad de color (8 bits para 256 colores) siguen las coordenadas de la imagen y la resolución, debiéndose tener cuidado con este concepto, ya que algunos programas guardan aquí la anchura y altura de la imagen. La siguiente paleta proviene de las épocas de imágenes en 16 colores y no tiene ninguna importancia en el caso de 256 colores, pero como medida de seguridad deberían colocarse aquí los primeros 16 colores de la paleta.

En el *Offset 65* sigue el número de planos de bit utilizados (1 para 256 colores) y a continuación la cantidad de bytes que existen en una línea de imagen. Este último valor está redondeado a un valor entero. Del siguiente WORD *Offset 68* se puede extraer si la paleta contiene colores o niveles de gris (1 para color o blanco y negro). A este valor solamente le siguen bytes de relleno.

Los datos de la paleta en sí se encuentran en las imágenes de 256 colores al final del archivo y están especificados por la marca *0ch*, por lo que se añaden 769 bytes más a los datos del gráfico en sí.

A continuación de la cabecera siguen directamente los datos gráficos, que pueden estar sin comprimir o empaquetados según el procedimiento RLE. Este sistema de compresión junta dos bytes iguales y seguidos en una combinación de 2 bytes. Los bytes individuales son escritos directamente al archivo y las repeticiones se identifican colocando los primeros dos bits superiores del byte. En este caso se trata de un byte de longitud, al cual sigue directamente un byte de datos. Este deberá ser escrito durante la descompresión tantas veces a la memoria de vídeo como indica el byte de longitud.

Por lo visto no se pueden evaluar de esta forma bytes individuales que tengan puestos los dos bits superiores, de forma que deberán ser guardados como repeticiones de byte de longitud 1. Dado que para ello se necesitan 2 bytes en el archivo comprimido, se trata más de un incremento que de una reducción de la cantidad de datos. Si la imagen tiene pocos puntos de un solo color, lo que suele pasar a menudo con imágenes de 256 colores, se suele dar el caso que el archivo comprimido sea más grande que el mismo sin comprimir.

La gran ventaja de este método de compresión es la gran velocidad de procesamiento: las simples repeticiones de byte se puede descodificar de forma mucho más rápida y simple que los códigos LZW del formato GIF, que tienen que ser buscados siempre en el alfabeto.

El objetivo de este capítulo es crear un capturador de pantallas universal (copia el contenido de la pantalla a un archivo) para ambos formatos 320 x 200 x 256 que pueda ser tratado con efectos especiales como división de pantalla (Split-Screen). Excepcionalmente no se va a insistir en la máxima velocidad, sino en la facilidad de comprensión, por lo que el compresor ha sido escrito en Pascal. Su intención es demostrar cómo se trabaja con imágenes PCX, es decir, cómo se almacenan.

Una posible aplicación de este programa sería capturar imágenes de una demo para guardarlas junto con los nombres del programador, grafista y músico. El código fuente de este programa está situado en el archivo GRABBER.PAS:

```
{$m 1024,0,0}                    {poco Stack y ningún Heap}
Uses ModeXLib,Crt,Dos;

Var OldInt9:Pointer;             {ptro. controlador teclado anterior}
    active:Boolean;              {activo si ya hay hardcopy en marcha}
    nr:Word;                     {número de imagen, para dar nombres}
    instalado:Boolean;           {¿ya instalado?}

    Mode,                        {modo actual VGA: 13h, ffh (Mode X)}
                                 {o 0 (ninguno de los 2}
    Split_at,                    {Split-Line (línea gráfica)}
    LSA,                         {Linear Starting Address}
    Skip:Word;                   {nº de bytes a saltar}
Procedure GetMode;
{determina modo gráfico actual 13h o Mode X (Nr. 255)}
{y datos de marco (Split-Line, direcc. inicial)}
Begin
  Mode:=$13;                     {Mode 13h Standard}
  asm                            {determinar modo BIOS}
    mov ax,0f00h                 {función: Video-Info}
    int 10h
    cmp al,13h                   {modo BIOS 13h activado?}
    je @Bios_ok
    mov mode,0                   {si no -> ni modo 13 ni X activos}
  @bios_ok:
  End;
  If Mode=0 Then Exit;           {modo erróneo -> cancelar}

  Port[$3c4]:=4;                 {leer TS-Register 4 (Memory Mode)}
  If Port[$3c5] and 8 = 0 Then   {Chain 4 (Bit 3) ¿inactivo?}
    Mode:=$ff;                   {entonces modo X}

  Port[$3d4]:=$0d;               {Linear Starting Address Low, 0dh}
  LSA:=Port[$3d5];               {leer}
  Port[$3d4]:=$0c;               {Linear Starting Address High, 0ch}
  LSA:=LSA or Port[$3d5] shl 8;  {leer y registrar}

  Port[$3d4]:=$18;               {Line Compare CRTC 18h}
  Split_at:=Port[$3d5];          {leer}
  Port[$3d4]:=7;                 {Overflow Low}
  Split_at:=Split_at or          {desenmascarar bit 4 y desplazarlo a
bit 8}
    (Port[$3d5] and 16) shl 4;
  Port[$3d4]:=9;                 {Maximum Row Address}
  Split_at:=Split_at or          {desenmascarar bit 6, despl. a bit 9}
    (Port[$3d5] and 64) shl 3;
  Split_at:=Split_at shr 1;      {convertir en líneas de pantalla}

  Port[$3d4]:=$13;               {Row Offset (CRTC Register 13h)}
```

```
  Skip:=Port[$3d5];                {leer}
  Skip:=Skip*2-80                  {leer diferencia con la distancia
"normal" de línea}
End;

Procedure PCXShift;assembler;
{prepara paleta actual a PCX (desplazar 2 a la izq.)}
asm
  mov si,offset palette            {puntero a paleta en ds:si}
  mov cx,768                       {procesar 768 bytes}
@lp:
  lodsb                            {obtener valor}
  shl al,2                         {desplazar}
  mov ds:[si-1],al                 {devolver a la posición antigua}
  loop @lp                         {y terminar bucle}
End;

Var pcx:File;                      {archivo PCX en el disco}

Procedure Hardcopy(DirecIni,splt:Word;s : string);
{copia gráfico 320x200 (Modo 13 o X) como PCX en archivo}
{inicio actual de pantalla (Linear Starting Address) en DirecIni}
{línea split en splt}
Var Buf:Array[0..57] of Byte;      {acoge datos antes de guardar}
    Aux_Ofs:Word;
const
  Header1:Array[0..15] of Byte     {cabecera PCX, 1ª parte}
    =($0a,5,1,8, 0,0, 0,0, $3f,1, 199,0,$40,1,200,0);
  Header2:Array[0..5] of Byte      {cabecera PCX, 1ª parte}
    =(0,1,$40,1,0,0);
  plane:Byte=0;                    {nº de plane actual}

var count:Byte;                    {nº de caracteres iguales}
    Valor,                         {valor obtenido}
    lastbyt:Byte;                  {valor anterior}
    i:word;                        {contador de bytes}

begin
asm                                {leer paleta}
  xor al,al                        {comenzar en color 0}
  mov dx,3c7h                      {al DAC con Pixel Read Address}
  out dx,al

  push ds                          {puntero es:di a paleta}
  pop es
  mov di,offset palette
  mov cx,768                       {leer 768 bytes}
  mov dx,3c9h                      {Pixel Color Value}
  rep insb                         {y leer}
```

```
    cmp mode,13h                {modo X ?}
    je @Linear                  {entonces:}
    mov dx,03ceh                {fijar modo de lectura/escritura 0}
    mov ax,4005h                {mediante registro GDC 5 (GDC Mode)}
    out dx,ax
  @Linear:
  End;

    Assign(pcx,s);              {abrir archivo para escribir}
    Rewrite(pcx,1);
    BlockWrite(pcx,Header1,16); {escribir parte 1 cabecera}
    PCXShift;                   {preparar paleta}
    BlockWrite(pcx,palette,48); {colocar primeros 16 colores}
    BlockWrite(pcx,Header2,6);  {escribir parte 1 cabecera}
    FillChar(buf,58,0);         {escribir 58 ceros (llenar cabecera)}
    BlockWrite(pcx,buf,58);
    plane:=0;                   {comenzar con plane 0}
    count:=1;                   {inicializar con 1}
    If splt<200 Then
      If Mode = $ff Then
        splt:=splt*80 Else      {calcular Split-Offset}
        splt:=splt*320 Else     {diferente, según modo}
      Splt:=$ffff;
    If Mode=$13 Then            {LSA se refiere al modo de planes!}
      DirecIni:=DirecIni*4;
    for i:=0 to 64000 do Begin  {editar cada punto}
    If i shr 2 < splt Then
    aux_ofs:=(i div 320) * skip {fijar offset auxiliar teniendo en}
                                {cuenta la anchura de línea}
    Else
    aux_ofs:=((i shr 2 - splt) div 320) * skip;
                                {durante Splitting ref. a VGA-Start}
  asm                           {leer punto}
    mov ax,0a000h               {cargar segmento}
    mov es,ax
    mov si,i                    {cargar offset}
    cmp mode,13h                {modo 13h ?}
    je @Linear1
    shr si,2                    {no, calcular offset}
  @Linear1:
    cmp si,splt                 {alcanzada Split-Line?}
    jb @seguir                  {no, seguir}
    sub si,splt                 {sino, referir lo demás al}
    sub si,DirecIni             {inicio de pantalla}
  @seguir:
    add si,DirecIni             {sumar dirección de inicio}
    add si,aux_ofs              {sumar offset auxiliar}
```

```
  cmp mode,13h                    {¿modo 13h?}
  je @Linear2                     {no, método lectura modo X}
  mov dx,03ceh                    {seleccionar plane actual mediante}
  mov ah,plane                    {GDC-Register 4, Read Plane Select}
  inc plane                       {y seguir}
  mov al,4
  and ah,03h
  out dx,ax
@Linear2:
  mov al,es:[si]                  {leer byte}
  mov Valor,al                    {y guardar en variable Valor}
End;
  If i<>0 Then Begin              {no hay compresión en el 1º byte}
  If (Valor = lastbyt) Then Begin {¿bytes iguales?}
    Inc(Count);                   {incrementar contador}
    If (Count=64) or              {¿contador muy alto?}
     (i mod 320 =0)  Then Begin   {¿o inicio de línea?}
      buf[0]:=$c0 or (count-1);   {guardar temporalmente}
      buf[1]:=lastbyt;            {escribir valor/estado de contador}
      count:=1;                   {reinicializar contador}
      BlockWrite(pcx,buf,2);      {y al disco}
    End;
  End Else                        {diferentes bytes:}
    If (Count > 1) or             {¿eran varios iguales?}
    (lastbyt and $c0 <> 0) Then   {¿es grande para escribir directo?}
    Begin
      buf[0]:=$c0 or count;       {cantidad y valor al disco}
      buf[1]:=lastbyt;
      lastbyt:=Valor;             {valor actual para compresión}
      Count:=1;                   {guardar y reinicializar}
      BlockWrite(pcx,buf,2);
    End Else Begin                {único byte legal}
      buf[0]:=lastbyt;            {escribir directamente}
      lastbyt:=Valor;             {guardar valor actual para después}
      BlockWrite(pcx,buf,1);
    End;

  End Else lastbyt:=Valor;        {el primer byte sólo guardar}
End;
  buf[0]:=$0c;                    {insertar denominación paleta}
  blockwrite(pcx,buf[0],1);       {y escribir}
  blockwrite(pcx,palette,256*3);  {añadir paleta}
  Close(pcx);                     {cerrar archivo}
End;


Procedure Action;
{se llama al activar el Hot-Key}
Var nrs:String;                   {cadena para nombre}
```

```
Begin
  if not active Then Begin        {sólo si no activo}
    active:=true;                 {marcar como activo}
    str(nr,nrs);                  {convertir y aumentar cadena}
    Inc(nr);
    GetMode;                      {obtener modo gráfico, etc.}
    If Mode <> 0 Then
      HardCopy(LSA,Split_at,'hard'+nrs+'.pcx');
                                  {realizar hardcopy}
    active:=false;                {liberar nueva activación}
  End;
End;

Procedure Handler9;interrupt;assembler;
{nuevo controlador de interr. para IRQ de teclado}
asm
   pushf
   call [oldint9]                 {llamar antiguo control. IRQ 1}

  cli                             {no hay más interrupciones}
  in al,60h                       {leer Scancode}
  cmp al,34d                      {¿G?}
  jne @listo                      {no -> terminar controlador}
  xor ax,ax                       {cargar segmento 0}
  mov es,ax
  mov al,es:[417h]                {leer estado de teclado}
  test al,8                       {Bit 8 (tecla Alt) activo?}
  je @listo                       {no -> terminar controlador}

  call action                     {realizar hardcopy}
@listo:
  sti                             {permitir interrupciones}
End;

Procedure identific;assembler;
{procedimiento dummy, contiene Copyright para reconocimiento de ins-
talación. ¡NO ES CÓDIGO EJECUTABLE!}
asm
  db 'Screen-Grabber, (c) Data Becker 1994';
End;

Procedure Check_Inst;assembler;
{comprueba si Grabber ya está instalado}
asm
  mov instalado,1                 {suposición: ya instalado}
  push ds                         {¡ds aún se necesita!}
  les di,oldint9                  {cargar ptro. a controlador anterior}
  mov di,offset identific         {mismo segmento proc. identific}
  mov ax,cs                       {ds:si a etiqueta de este programa}
```

```
  mov ds,ax
  mov si,offset identific
  mov cx,20                    {comparar 20 caracteres}
  repe cmpsb
  pop ds                       {restaurar ds}
  jcxz @instalado              {igual, ya instalado}
  mov instalado,0              {no instalado: guardar}
@instalado:
End;

Begin
  nr:=0;                       {1er nombre de archivo: hard0.pcx}
  GetIntVec(9,OldInt9);        {obtener vector interrupción antiguo}
  Check_Inst;                  {comprobar si ya instalado}
  If not instalado Then Begin  {si no:}
    SetIntVec(9,@Handler9);    {instalar nuevo controlador}
    WriteLn('Grabber instalado');
    WriteLn('(c) Data Becker 1994');
    WriteLn('Activación con <ALT> G');
    Keep(0);                   {mostrar mensaje; terminar residente}
  End;
  WriteLn('Grabber ya está instalado');
                               {si ya instalado, mensaje y final}
End.
```

El programa principal es típico de los TSR, aunque sea hayan omitido cosas como la desinstalación a causa de las capacidades limitadas del Pascal. Después de comprobar si el programa ya ha sido instalado con anterioridad, se desvía en el caso de un resultado negativo el vector de interrupción del teclado, se emite un mensaje y se abandona el programa manteniéndolo residente en memoria. De lo contrario, se le comunica al usuario que ya existe una copia del programa en memoria, y se finaliza de forma normal. La comprobación se efectúa mediante una simple comparación de un procedimiento (identific) por un lado en el programa cargado y por otro en el controlador actual del teclado.

El controlador, que está enganchado en la interrupción del teclado (IRQ 1, o sea interrupción 9), tiene ahora la tarea de traspasar todas las pulsaciones del teclado al controlador antiguo, a fin de garantizar un funcionamiento correcto del sistema, y activa además, al pulsarse la combinación de teclas [Alt]+[G], el capturador. Para ello se compara cada código de escáner con el código de la tecla [G] y en caso de coincidencia se comprueba además a través del flag de teclado del BIOS en la dirección *00417h* si se ha pulsado también la tecla [Alt]; en el caso de que se cumplan las dos condiciones se activa el procedimiento *Action*.

Lo primero que hace este procedimiento es asegurarse que durante la grabación de la imagen no se ha vuelto a pulsar la combinación de teclas de activación (Hotkey), lo cual causaría complicaciones. Se determina ahora el nombre del archivo en base a un número correlativo, por si se efectúan varias capturas de panta-

lla. Antes de activarse el procedimiento de captura de pantalla en sí, hay que determinar el modo gráfico actual, ya que el capturador ha sido diseñado para 320 x 200 puntos.

Esta tarea es realizada por el procedimiento *GetMode*: primero se determina el modo del BIOS, que en ambos casos es *13h*. Si el BIOS devuelve un valor diferente, la impresión de pantalla no se podrá efectuar, y el procedimiento se finaliza con un *0* como modo. A través del *Chain4-Bit* del registro *TS 4* se diferencia ahora entre el *Mode 13h y Mode X*.

Dado que se pretende que el capturador genere una imagen lo más real posible aunque existan diversos efectos, aún se deberán leer algunos registros CRTC. Primero se comprueba el *Line Starting Adress*, que se utiliza para el desplazamiento de pantalla, a continuación el registro *Line-Compare* para la línea de división, Split line (este registro está repartido entre 3 registros CRTC, que aquí se vuelven a juntar), y finalmente el *Row Offset*, que se utiliza en resoluciones virtuales mayores (640 x 400). Gracias a este Row Offset se calcula ahora la anchura de salto *Skip* entre dos líneas, cuyo valor en resoluciones normales de 320 x 200 es de 0 y en una resolución virtual de 640 x 400 de 80.

A continuación se arranca el procedimiento de impresión o captura de pantalla mediante *Action*, al cual se le mandan como parámetros los valores CRTC determinados así como el nombre del archivo. Este procedimiento lee primero la paleta actual desde el *DAC* y activa el modo de lectura 0, si está activo el *Mode X*. La paleta se convierte a un formato de 8 bits mediante el procedimiento *PCXShift*, realizándose el camino inverso de la lectura de un archivo GIF: cada valor se desplaza en 2 bits hacia la izquierda.

El encabezamiento ya preparado junto con los primeros 16 registros de la paleta es escrito al archivo y rellenado hasta los 128 bytes. Aquí aún se calculan algunas variables: *Split* se envía como línea y tiene que ser reconvertido en un offset, ya que el bucle de copiado trabaja en base a un Offset. En el *Mode 13h* la dirección de inicio debe ser multiplicada por cuatro, a fin de tener en cuenta la estructura especial (*Chain 4)* de este modo.

En el bucle principal (*i*) se calcula un nuevo offset (*aux_ofs)* que se aplica en el caso de imágenes demasiado anchas (por ejemplo en desplazamientos tipo scrolling en todas las direcciones). Si el offset actual está situado antes de la *Split-Line*, simplemente se multiplica el número de líneas evaluadas por la distancia por línea. Después de la línea de división (*Split Line)* también hay que tenerla en cuenta, a fin de obtener una referencia al inicio de la memoria.

El siguiente bloque *ASM* sirve para leer un byte de la memoria de vídeo. Primero se carga *ES:SI* con un puntero sobre el pixel actual. En el modo X, tal como se ha aplicado antes, se divide el offset entre cuatro.

Si la línea de división (*Split Line)* ya se ha sobrepasado se restablece la referencia al inicio de la memoria de vídeo, desactivando el efecto de la dirección de inicio modificada (mediante una substracción) y calculando la distancia hasta la línea de división (*Split Line),* también mediante una substracción. En el *Mode X* se determina y establece el plano actual. A continuación se lee el pixel a la variable *Valor* y a continuación se comprime.

El primer byte tiene que ser cargado una vez para poder ser tratado en la siguiente pasada como Last-Byte (último byte). Cada siguiente byte es comparado con el anterior y en caso de coincidencia solamente se incrementa el contador *count*. Un caso especial se produce cuando el contador supera el límite de 63, que es el valor máximo que se puede guardar en un byte de longitud. En este caso se vacía el contador del archivo y se comienza a contar de nuevo. Otro caso especial es el final de línea. Originalmente el formato PCX trabaja basado puramente en líneas, lo que repercutía en la compresión: las repeticiones de caracteres no podrían sobrepasar un fin de línea. Dado que algunos programas de dibujo antiguos aún mantienen este aspecto, deberá tenerse en cuenta durante la compresión.

Si el carácter no se corresponde con el anterior, se ejecuta la siguiente orden *Else.* Aquí se comprueba si se trata de una repetición (*count > 1)* o si el carácter ha utilizado los dos bits superiores. En ambos casos habrá que escribir un carácter comprimido con byte contador (en el segundo caso 1) y byte de datos. *Lastbyt* se actualiza al nuevo valor, a fin de poder comprimir en la siguiente pasada. Si el *Count* en cambio es 1 y el Last-Byte uno legal (bits superiores vacíos) se salta al bloque *else* de esta estructura *If,* desde donde se escribe este byte directamente al archivo sin comprimirlo y se pone igualmente el *Lastbyt* en su valor actual.

Después de haber escrito de este modo toda la imagen al archivo solamente falta añadir la paleta (con la identificación *0h)* y cerrar el archivo. A continuación también se finaliza el controlador.

Como ya hemos dicho antes, este programa no pretende ser un ejemplo de velocidad, pero en el caso de los capturadores de pantalla tampoco se trata de esto. Lo importante es explicar la estructura de un archivo PCX mediante un ejemplo (aquí uno de escritura en vez de uno de lectura).

El programa se queda residente en memoria después de su activación a la espera de la combinación de teclas (Hotkey); no existe ninguna otra manera de eliminarlo que un Reset. La única forma de crear un programa TSR confortable que incluya desinstalación es utilizando ensamblador, pero por lo menos se puede evitar con éxito en Pascal una nueva instalación, tal como muestra el procedimiento *Check_Inst.*

4.4 VGA hasta el último bit

En la continuación de este libro vamos a conocer una gran cantidad de efectos gráficos que se basan en su mayoría en manipulaciones de registros. Estos registros ofrecen una cantidad inagotable de posibilidades de influir hasta en el último detalle de las características de la VGA, habiendo quedado hasta ahora muchos registros sin utilizar ya que aparentemente parecían inútiles. Pero utilizando un poco la fantasía se pueden extraer efectos interesantes de casi todos los registros, lo único que hace falta es experimentar un poco. Para ayudarle un poco y aclarar los efectos que describiremos más adelante, este capítulo contiene todos los registros de la VGA hasta el último bit.

En principio se puede experimentar libremente con los valores, pero se debe tener cuidado con los registros de *Timing* (registros CRTC *0-7)* ya que una utilización continuada con valores poco apropiados (el monitor empieza a pitar de forma continuada) puede dañar el monitor.

Pueden existir personas que aprovechen estas características de forma negativa, ya que con ellas es posible dañar los monitores. Por esta razón le recomendamos nuevamente CUIDADO y apagar inmediatamente el monitor cuando esté comience a pitar. Si se reacciona a tiempo se podrá evitar un daño irrecuperable en el monitor. Normalmente se necesitan unos cuantos segundos antes de que el monitor se despida de forma definitiva. Cuando realice experimentos de este tipo deberá tener siempre un dedo en el interruptor de encendido del monitor.

Lo peor que pueden conseguir todos los demás registros es que el monitor se bloquee. Los registros se pueden dividir en dos grandes grupos: primero los registros individuales, a los cuales se puede acceder directamente a través de su propia dirección de puerto, y por otro lado los registros indexados. Esto no tiene nada que ver con un autocontrol voluntario, *indexados* significa más bien que los registros no posee una dirección de puerto propia. Son componentes de otros componentes de la VGA y se seleccionan a través de una dirección de puerto fija y se modifican a través de otra.

Miscellaneous Output Register (registros de salida misceláneos)
Lectura: Puerto 3cch; escritura: 3c2h

Bit	Significado	Acceso
7	polaridad vertical del Retrace	RW
6	polaridad horizontal (con bit 7 resolución vertical)	RW
5	Selección de página (page Select) para direccionamiento par/impar	RW
4	reservado	
3,2	Selección de reloj (Clock select) (resolución horizontal)	RW
1	Enable RAM), 1= posible el acceso a RAM desde la CPU	RW
0	Dirección E/S (I/O), 1= monocromo (3bxh), 0= Color (3dxh)	RW

Explicación:

Bit 7,6: A través de la polaridad de las dos señales de Retrace se determina la resolución física vertical: *00b* está reservado, *01b* significa 350 caracteres, *10b* 400 líneas de punto (también 320 x 200, ver registro CRTC *9*), *11b* 480 líneas.

Bit 5: Indica en el modo Par/impar el Bit 0, lo que significa que en todos los planos se ocupan o bien todas las direcciones pares (Bit 5= 0) o impares (Bit 5 = 1), ver registro TS 4.

Bit 3,2: Determina a través de la frecuencia de píxeles la resolución horizontal: *00b* significa 640 puntos horizontales, *01b* 720 puntos, *02b* 800 puntos y *11b* está reservado.

Bit 1: Regula el acceso a la RAM de la VGA por parte de la CPU (0= sin acceso).

Bit 0: Con este bit se determina en que dirección de puerto se encuentran el CRTC (*3 x 4h / 3 x 5h), Input Status Register 1(3 x ah)*. Si este bit contiene un 1 habrá que sustituir la *x* con una *d*, con un *0* con una *b*.

Input Status Register Lectura: puerto 3c2h		
Bit	**Significado**	**Acceso**
7	Interrupción (Interrupt) CRT	RO
-0	específico del fabricante, suele ser Feature Code etc.	RO

Explicación:

Bit 7: Muestra un retrace vertical, siempre que se haya activado el Interrupt de Retrace (normalmente está desactivado mediante un interruptor DIP en el hardware). Se borra a través del registro CRTC 4.

```
Input Status Register 1
Port 3dah (color) o 3bah (monocromo), según el Bit 0
Miscellaneous Output Register
```

Bit	Significado	Acceso
7,6	reservado (parcialmente Bit 3 invertido)	
5,4	bits de prueba, variables según tarjeta gráfica	RO
3	retrace vertical	RO
2,1	reservados	
0	Display enable complement	RO

Explicación:

Bit 3: Un valor de *1* indica que se está produciendo un Retrace vertical. Mediante este bit se debería sincronizar la construcción de la imagen, a fin de evitar que la construcción y modificación de la imagen coincidan, lo cual produce interferencias en la imagen (ver apartado 4.1)

Bit 0: Señal *Display enable* invertida, con 1 significa que se está ejecutando un Retrace vertical u horizontal. Este Bit se puede utilizar para determinar la línea dibujada en este momento; después de un Retrace vertical (bit 3) un paso de este bit de 1 a 0 significa el comienzo de una nueva línea; una aplicación son por ejemplo las conocidas Barras Copper (apartado 6.8)

Además existen, según el fabricante, otros registros que no están normalizados y que reflejan determinadas características (por ejemplo, Feature Connector) de la correspondiente tarjeta. Por esta razón no resultan apropiados para una programación general.

El controlador del tubo de rayos catódicos (CRTC)

El chip de la VGA que contiene más registros es el controlador del tubo de rayos catódicos (Cathod Ray Tube controller, CRTC), que sirve para generar la señal de vídeo. El CRTC es programable con bastante libertad y la libertad de movimiento del rayo catódico puede ser modificada ampliamente, algo que podría causar ciertos problemas a su monitor, por lo que habrá que andar con sumo cuidado. Por esta razón estos registros están protegidos contra una eventual escritura involuntaria mediante un bit de protección (*Protection Bit, Registro CRTC 11).* Pero el CRTC posee bastantes más registros que no tienen nada que ver con el timing del rayo (por ejemplo, el *Linear Starting Adress).* Estos registros sí que son manipulables libremente.

A estos registros se accede a través de los dos direcciones de puerto: *3d4h* sirve como registro de índice y el puerto *3d5h* como registro de datos (en el caso de representación en monocromo - bit 0 del *Miscellaneous Output Register* = 0 - son los puertos *3b4h/3b5h).* Se accede a un registro específico escribiendo su número al registro de índice y accediendo posteriormente con escritura o lectura al registro de datos, en el cual se tendrá a disposición el correspondiente registro CRTC. Después de haber situado el índice se podrá acceder al registro todas las veces que se desee, manteniéndose el valor del índice. Por razones de velocidad se puede ejecutar un único acceso de lectura mediante una salida WORD al puerto de índice, debiendo contener el Low-Byte el número de registro y el High-Byte el valor de registro.

Registro CRTC 0: Horizontal Total		
Bit	**Significado**	**Acceso**
7-0	Cantidad de caracteres por línea (-5 en VGA)	RW

Explicación:

Bit 7-0: Este registro indica la longitud total de línea en unidades de carácter (*Character Times Units*). Una unidad de carácter corresponde o bien a 8 puntos (Registro TS 1, bit 0 = 1, por ejemplo, modos 320 x 200) o nueve puntos (Registro TS 1, bit 0= 0, por ejemplo, modo texto 3). El valor real para este registro deber ser reducido en cinco (dos en modos EGA)

Registro CRTC 1: Horizontal Display End		
Bit	**Significado**	**Acceso**
7-0	Final de la señal Display enable en caracteres	RW

Explicación:

Bit 7-0: Este registro indica en la práctica la cantidad de caracteres visibles (ocho o nueve puntos, ver registro 0)

Registro CRTC 2: Horizontal Blank Start		
Bit	**Significado**	**Acceso**
7-0	Inicio del periodo horizontal de borrado (blanking)	RW

Explicación:

Bit 7-0: Aquí se determina la posición en la que el CRTC desconecta el rayo catódico durante la construcción de las líneas. El periodo de borrado (blanking) contiene el *periodo de Retrace* y coloca el marco izquierdo y derecho de la imagen en negro.

Registro CRTC 3: Horizontal Blank End		
Bit	**Significado**	**Acceso**
7	Bit de prueba (uso normal : 1)	RW
6-5	Display Enable Skew (ralentización)	RW
4-0	Horizontal Blank End (Bit 4-0)	RW
6-5	Estos bits indican el número de caracteres que el CRTC "previsualiza" en la memoria (la señal de Display enable se ralentiza de forma correspondiente), a fin de poder ofrecer el siguiente carácter a tiempo, en la VGA normalmente 0.	

Explicación:

Bit 4-0: Los 5 bits inferiores del final de 6 bits del periodo de borrado se coloca aquí. El bit 5 de este valor se encuentra en el registro CRTC 5 bit 7.

Registro CRTC 4: Horizontal Sync Start		
Bit	**Significado**	**Acceso**
7-0	Posición de inicio del Retrace horizontal	RW
Bit 7-0	Este registro indica la posición (en caracteres) en la que debe comenzar el Retrace horizontal	

Registro CRTC 5: Horizontal Sync End		
Bit	**Significado**	**Acceso**
7	Bit 5 del Horizontal Blank End (registro 3)	RW
6-5	Horizontal Sync Skew (ralentización)	RW
4-0	Final del Retrace horizontal	RW
Bit 6-5	También se puede incluir una ralentización o pausa después de un Retrace horizontal. Ver registro 3, bit 6-5, en VGA normalmente 0.	

Explicación:

Bit 4-0: Estos bits definen el final de un Retrace horizontal, pero relativo al inicio; la primera vez que el contador de caracteres interno coincide en el bit 5 inferior con este registro, el Retrace ha finalizado.

Registro CRTC 6: Vertical total		
Bit	**Significado**	**Acceso**
7-0	Número de líneas total por imagen -2 (bits 7-0)	RW

Explicación:

Bit 7-0: Aquí se indica la altura total de la imagen en líneas (reducidas en 2, en el EGA en 1). La altura total es un valor de 10 bits (en la SuperVGA 11 bits), los bits 9-8 se encuentran en el registro *Overflow* (registro 7)

Registro CRTC 7: Overflow		
Bit	**Significado**	**Acceso**
7	Vertical Sync Start - bit 9	RW
6	Vertical Display enable End - bit 9	RW
5	Vertical Total - bit 9	RW
4	Line compare (Spli-Screen) - bit 8	RW
3	Vertical Blank Start - bit 8	RW
2	Vertical Sync Start - bit 8	RW
1	Vertical Display Enable End - bit 9	RW
0	Vertical Total - bit 8	RW

Explicación:

Bit 7-0 Este registro contiene los bits 8 y 9 de la mayoría de registros verticales, bits que no cabían en sus registros originales.

Registro CRTC 8: Initial Row Adress (dirección de fila inicial)		
Bit	**Significado**	**Acceso**
7	reservado	
6-5	Byte Panning	RW
4-0	Initial Row Address	RW

Explicación:

Bit 6-5 Mediante este registro se puede desplazar el contenido de la pantalla hasta en 3 bytes hacia la izquierda (¡en el *Modo X* hasta cuatro píxeles!); pero este desplazamiento se puede conseguir de forma mucho más efectiva mediante la *Linear Starting Adress* (Registro *0ch/0dh*).

Bit 4-0 Este registro indica al CRTC en que línea deberá comenzar después de un retrace vertical, normalmente la *0*. Si se incrementa este valor, el CRTC comenzará en una línea situada más abajo, o sea que desplazará el contenido de la pantalla hacia arriba. Este registro funciona de la misma manera en el modo texto, de forma que con él se puede realizar un desplazamiento vertical suave (*Smooth Scrolling)* (ver apartado 6.6)

Otra posible aplicación hace referencia al modo 320 x 200. Si se desea ralentizar el scrolling vertical manteniendo, a pesar de ello, 70 desplazamientos por segundo, a fin de evitar saltos, se puede actuar de la siguiente manera: primero se incrementa la dirección de inicio solamente en cada segunda pasada, a fin de reducir la velocidad; el scrolling fino se consigue después con este registro, que se hace pendular entre *0* y *1*. De esta forma se consigue un scrolling de una línea (= media línea en el modo 320 x 200 !).

Registro CRTC 9: Maximum Row Adress (dirección de fila máxima)		
Bit	**Significado**	**Acceso**
7	Double Scan (duplicación de líneas)	RW
6	Line Compare (split screen - registro 18h) bit 9	RW
5	Vertical Blank Start bit 9	RW
4-0	Cantidad de líneas por líneas de caracteres -1	RW

Explicación:

Bit 7 Si se coloca este bit en *1*, se divide el ratio de reloj (*Clock-Rate)* entre dos, representándose cada línea dos veces. Esto estaba previsto para la generación de los modos de 200 líneas en un resolución física de 400 líneas. La mayoría de BIOS lo consiguen situando el bit *0*.

Bit 4-0 Indica la altura de los caracteres en el modo texto reducida en 1 (15 en el modo VGA 3,9 x 16 puntos por carácter). También se puede utilizar en el modo gráfico, a fin de reducir la resolución vertical (representación múltiple de cada línea, cuando bit 4-0 > *0)*; ver apartado 6.7

Registro CRTC 0ah: Línea de inicio del cursor (Cursor Start Line)		
Bit	**Significado**	**Acceso**
7-6	reservado	
5	Cursor activo (0) / cursor apagado (1)	RW
4-0	Línea de inicio	RW

Explicación:

Bit 4-0 Línea dentro del carácter, en la cual debe comenzar la representación del cursor.

Registro CRTC 0Bh: Línea de fin del cursor (Cursor End Line)		
Bit	**Significado**	**Acceso**
7	reservado	
6-5	Cursor skew	RW
4-0	Línea de final	RW

Explicación:

Bit 6-5 También se puede incluir una ralentización al representar el cursor, a fin de garantizar la representación en los bordes izquierdo y derecho, en la VGA normalmente *0*.

Bit 4-0 Línea dentro del carácter, en la cual finaliza la representación del cursor.

Registro CRTC 0ch: Dirección de inicio de línea (Linear Starting Adress High)		
Bit	**Significado**	**Acceso**
7-0	Dirección de inicio de línea Bit 15-8	RW

Explicación:

Bit 7-0 Este registro coloca los bits 15-8 de la dirección de inicio de 16 bits. Esta dirección indica el offset dentro de la memoria de vídeo en la cual el CRTC comienza con la lectura de los datos de la imagen. Si se modifica se puede conseguir un scroll vertical y horizontal (ver apartado 6.3), pero en el modo texto el scroll solamente se efectúa línea a línea, realizándose el ajuste fino mediante el registro *ATC 13h (Horizontal Pixel Panning)* y *CRTC 8 (Initial Row Adress)*, ver capítulo 6.6. Lo importante es que la dirección real debe dividirse entre dos en el modo *Odd/Even (par/impar)* y entre cuatro en el modo *Chain 4 (p.ej. 13h)*, antes de poder escribirla a este registro.

Registro CRTC 0dh: Dirección de inicio de línea (Linear Starting Adress Low)		
Bit	**Significado**	**Acceso**
7-0	Dirección de inicio de línea bit 7-0	RW

Explicación:

Bit 7-0 Indican el *Word* de valor bajo de la dirección de inicio de la memoria, ver registro *0ch*.

Registro CRTC 0eh: Dirección de cursor (Cursor Adress High)		
Bit	**Significado**	**Acceso**
7-0	Posición del cursor como offset (bit 15-8)	RW

Explicación:

Bit 7-0 Indican la posición actual del cursor en la memoria de vídeo.

Registro CRTC 0fh: Dirección de cursor (Cursor Adress Low)		
Bit	**Significado**	**Acceso**
7-0	Posición del cursor como offset (bit 7-0)	RW

Explicación:

Bit 7-0 Low-Word de la dirección del cursor , ver registro *0eh*.

Registro CRTC 10h: Vertical Sync Start		
Bit	**Significado**	**Acceso**
7-0	Línea en la cual comienza el Retrace vertical	RW

Explicación:

Bit 7-0 Contienen los bit 7-0 de la línea de 10 bit, en la cual comienza el Retrace vertical. Los bits 8 y 9 se encuentran en el registro *Overflow*.

Registro CRTC 11h: Vertical Sync End		
Bit	**Significado**	**Acceso**
7	Bit de protección	RW
6	reservado	
5	Interrupción del retrace vertical encendido (1)	RW
4	Restablecer la interrupción del retrace vertical (0)	RW
3-0	Línea en la cual finaliza el retrace vertical	RW

Explicación:

Bit 7 El bit de protección sirve, cuando está colocado, para evitar accesos de escritura a los registros CRTC 0-7 excepto al bit 4 del registro de Overflow (registro 7). Este bit es colocado por casi todos los BIOS y deberá ser borrado para operaciones de timing.

Bit 5 Si este bit está borrado y la interrupción anterior restablecida por borrarse el bit 4, el CRTC provocará en el siguiente retrace vertical una IRQ 2. Por desgracia, la mayoría de las tarjetas VGA no provocan esta interrupción, ya sea porque no son capaces o porque han sido configuradas por el fabricante mediante interruptores DIP de tal forma que no produzcan una interrupción, a fin de garantizar la compatibilidad.

Bit 4 Después de cada interrupción exitosa deberá borrarse este bit, en caso contrario no se producirá ninguna interrupción nueva (ver bit 5).

Bit 3-0 El final del retrace vertical también se determina en relación a su inicio, ya que solamente se utilizan 4 bits para la comparación. La primera vez que el contador de líneas interno coincide en su bit 4 inferior con este valor se finaliza un retrace que esté en marcha.

Registro CRTC 12h: Vertical Display End		
Bit	**Significado**	**Acceso**
7-0	última línea representada (bit 7-0)	RW

Explicación:

Bit 7-0 Después de la línea con el número indicado aquí el CRTC desconecta el rayo; los bits 8-9 de este valor se encuentran en el registro *Overflow*.

Registro CRTC 13h: Row Offset (Offset de línea)		
Bit	**Significado**	**Acceso**
7-0	Offset entre dos líneas de pantalla	RW

Explicación:

Bit 7-0 Este registro determina la distancia entre dos líneas en la memoria, o sea su longitud. La unidad sobre que se basa depende del tipo de direccionamiento a memoria activo: en el modo Double-Word se cuenta en bloques de 8 bytes, lo que significa, por ejemplo, que en el Modo *13h* un valor de registro de 40 corresponde a una anchura real de 40 * 8 = 320 bytes. En un direccionamiento con Word la unidad es de 4 bytes (por ejemplo, modo de texto 3, contenido de registro 40, o sea 40 * 4 = 160 bytes). Mediante este registro también se puede ejecutar un scrolling horizontal, introduciendo, por ejemplo, un 80. En este caso las líneas comienzan con una doble distancia, con lo que se obtiene, "además" de la porción visible, partes invisibles de la memoria de vídeo. Si se desplaza el inicio de la pantalla en pocos bytes, se podrá efectuar un scrolling horizontal sobre la pantalla virtual (ver apartado 6.3)

Registro CRTC 14h: Underline Location (Situación del subrayado)		
Bit	**Significado**	**Acceso**
7	reservado	
6	Direccionamiento Double-Word	RW
5	Linear Adress Count by 4 (Cuenta de dirección de línea por 4)	RW
4-0	Línea en la cual comienza el subrayado	RW

Explicación:

Bit 6 Enciende el modo *Doubleword* si se coloca en 1; en este modo la dirección se rota en 2 bits hacia la izquierda antes de enviarla a la memoria de vídeo para un acceso de lectura. De esta forma se leen primero en la memoria los offset *0,4,8 etc.* y luego los offset *1,5,9, etc.* Este modo está activo por ejemplo en el modo *13h*.

Bit 5 Si este Bit está colocado, la frecuencia de los caracteres durante el acceso a la memoria de vídeo se divide entre cuatro; se utiliza normalmente con el modo *Doubleword*.

Bit 4-0 En los modos monocromo se determina a través de estos Bit en que línea el ATC efectuará el subrayado. Si se reduce este valor, los caracteres también se podrán tachar o subrayar por encima; si el registro sobrepasa la altura del carácter, se desconecta el subrayado.

Registro CRTC 15h: Vertical Blank Start (Inicio borrado vertical)		
Bit	**Significado**	**Acceso**
7-0	Vertical Blank Start Bit 7-0	RW

Explicación:

Bit 7-0 Este registro indica la posición vertical a partir de la cual el CRTC debe desconectar el rayo. Dentro de este periodo de borrado se ejecuta el retrace vertical. Los bits 8 y 9 de este registro se encuentran en el registro de *Overflow*.

Registro CRTC 16h: Vertical Blank End (final borrado vertical)		
Bit	**Significado**	**Acceso**
7-0	Línea en la cual finaliza el borrado vertical	RW

Explicación:

Bit 7-0 Este registro se refiere en forma relativa al inicio de borrado (Blank Start), ya que solamente se utilizan 8 bits. Cuando los 8 bits inferiores del contador interno de líneas se corresponden con este registro, el *Blanking* (Borrado) finaliza.

Registro CRTC 17h: CRTC Mode		
Bit	**Significado**	**Acceso**
7	Hold Control	RW
6	Word Mode (0) / Byte Mode (1)	RW
5	Ocupación alternativa para el bit de dirección 0	RW
4	reservado	
3	Linear adress count (contador de dirección de línea) por 2	RW
2	Line Counter (contador de línea) por 2	
1	Ocupación alternativa para el bit de dirección 14	RW
0	Ocupación alternativa para el bit de dirección 13	RW

Explicación:

Bit 7 Un valor de *0* detiene el timing horizontal y vertical

Bit 6 Un 0 indica que el modo *Word* está activo, un *1* el modo *Byte*; si el bit 6 del registro *14h* está colocado (*Doubleword-Mode*), este bit carece de importancia.

Bit 5 Si este bit está borrado, en el modo *Word* el bit 15 del bit 13 no se traspasa al bit 0 (rota); esto sirve en algunas tarjetas EGA con sólo 64 Kbytes para evitar el desbordamiento en el modo *Word*.

Bit 3 Un bit colocado reduce a la mitad la frecuencia del CRTC en la memoria de vídeo. Funciona de forma similar al bit 5 del registro *14h*.

Bit 2 Si se sitúa este bit en *1*, el contador de líneas solamente se incrementará con cada segunda línea, duplicándose todo el timing vertical.

Bit 1,0 Estos dos bits provocan al borrarse una reprogramación del conducto de dirección 14 o 13. El bit 0 se representa en este caso en el bit 13 y el bit 1 en el bit 14 (solamente cuando el bit 1 de este registro = 0). La consecuencia es que las direcciones impares se leen 8 Kbytes después de las direcciones pares, lo que quiere decir que un bloque se determina según los dos bits inferiores. Esto sirve para emular el controlador 6845, que accede de esta forma a la memoria de vídeo. En el caso de la CGA el Bit 0 se coloca en 0, en la emulación Hércules en bit 1.

Registro CRTC 18h: Line Compare (comparación de línea)		
Bit	**Significado**	**Acceso**
7-0	Line Compare Bit 7-0	RW

Explicación:

Bit 7-0 Indican la línea física en la cual el CRTC comienza a recuperar nuevamente sus datos desde el inicio de la memoria de vídeo. De esta forma se puede conseguir una pantalla dividida en dos: en la parte superior se representa la zona de memoria determinada por la *Linear Starting Adress*, en la parte inferior se muestra el inicio de la memoria de vídeo (ver apartado 6.2). El bit 8 de este registro se encuentra en el *Overflow* y el bit 9 en el registro de *Maximum Row Adress* (Dirección de línea máxima).

Timing Sequenzer (TS)

Las tareas del timing sequenzer se circunscriben principalmente a la administración de la memoria. Los accesos a esta se desvían según la configuración actual a determinados planos (planes) y se combinan con el juego de caracteres actual. Además, internamente el TS es responsable del refresco (refresh) de la memoria de vídeo.

Se accede al TS de la misma forma que al CRTC a través de un registro de índices (en el puerto *3c4h)*, pudiéndose situar aquí un registro a través de un simple *Word-Out*.

Registro TS 0: Reset síncrono		
Bit	**Significado**	**Acceso**
7-2	reservado	
1	reset síncrono	RW
0	reset asíncrono	RW

Explicación:

Bit 1 Si se desactiva este bit, el TS ejecuta un reset síncrono, es decir, restablece todos los registros y luego se desconecta, hasta que el Bit 0 y 1 vuelvan a estar colocados. Durante este tiempo también se desactiva el refresco de la RAM, de forma que el reset debería finalizarse lo más rápido posible, a fin de evitar pérdidas de datos. Este reset siempre se debería ejecutar cuando se vayan a modificar registros TS.

Bit 0 Si se desactiva este bit, se ejecuta un reset asíncrono, que tiene las mismas características que el reset síncrono, con la excepción que no se borra el registro de selección de fuente (*Font-Select)*, o sea que se mantiene el juego de caracteres actual.

Registro TS 1: Modo TS		
Bit	**Significado**	**Acceso**
7-6	reservado	RW
5	Screen off (apagar pantalla)	RW
4	Shift 4	RW
3	Dot Clocks (frecuencia de puntos) / 2	RW
2	Video load (carga de vídeo) / 2	RW
1	reservado	
0	estado TS	RW

Explicación:

Bit 5 Si se activa este bit, el TS desconecta la pantalla, permitiéndose de esta forma un acceso a la RAM más rápido por parte de la CPU.

Bit 4 Un bit activado indica que los latches se cargarán a una cuarta parte de su frecuencia.

Bit 3 En algunos modos 320 x 200 se coloca este bit para reducir la resolución horizontal de 640 a 320.

Bit 2 Un bit situado indica que los latches del ATC se cargarán con la mitad de su frecuencia.

Bit 0 El estado 0 sitúa la anchura de los caracteres en nueve puntos, un estado 1 en ocho puntos. Este valor es importante para el cálculo del timing horizontal, cuyos registros se basan todos en una unidad de carácter.

Registro TS 2: Write Plane Mask (máscara plano de escritura)		
Bit	**Significado**	**Acceso**
7-4	reservado	
3	Acceso de escritura al plano 3	RW
2	Acceso de escritura al plano 2	RW
1	Acceso de escritura al plano 1	RW
0	Acceso de escritura al plano 0	RW

Explicación:

Bit 3-0 Mediante este registro se puede incluir (1) o excluir (0) determinados planos para los accesos de escritura.

Registro TS 3: Font select (selección de fuente)		
Bit	**Significado**	**Acceso**
7-6	reservado	
5	Fuente B bit 2	RW
4	Fuente A bit 2	RW
3-2	Fuente B bits 0 y 1	RW
1-0	Fuente A bits 0 y 1	RW

Explicación:

Bit 5-0 Se indica la situación de la fuente (juego de caracteres) en la memoria, utilizándose la siguiente codificación:

Bits	Offset
000	0
001	16 Kbytes
010	32 Kbytes
011	48 Kbytes
100	8 Kbytes
101	24 Kbytes
110	40 Kbytes
111	56 Kbytes

La fuente A (font A) determina el aspecto de los caracteres, cuyo byte de atributo, bit 3, no está activado, la fuente B se activa cuando se coloca este bit.

Registro TS 4: Memory Mode (modo de memoria)		
Bit	**Significado**	**Acceso**
7-4	reservado	
3	Chain 4	RW
2	Modo par /impar (Odd/even)	RW
1	Memoria extendida	RW
0	reservado	

Explicación:

Bit 3 Activa el modo *Chain* 4 (1), de forma que las conducciones de dirección 0 y 1 se utilizan para la selección del plano, tanto en accesos de la CPU de lectura como de escritura. Este modo se utiliza en el modo *13h*. Si se desactiva y además se sitúa el CRTC en el modo *Byte*, ya se estará en el modo X. (Ver apartado 5.2)

Bit 2 Si este bit está desactivado, estará activo el modo *par/impar (odd/even)*. Este funciona de forma similar al modo *Chain 4*: El bit 0 de las conducciones de memoria se utiliza aquí para la selección de planos pares o impares. En el modo texto esto significa por ejemplo que los códigos ASCII de los caracteres se colocan en los planos 0 y 2, y en cambio los bytes de atributo en los planos 1 y 3. Debería coincidir siempre con el bit 4 del registro GDC 5 (¡bit invertido!)

Bit 1 Este bit indica la memoria instalada: 0 significa 64 Kbytes de memoria de vídeo, un 1 corresponde a 256 Kbytes. Solamente se podrá utilizar el bit 2 para la selección de fuente cuando esté activo este bit.

Graphics Data Controller (GDC)

Este componente controla los accesos de la CPU a la memoria a un nivel superior que el TS. Regula las manipulaciones que sufre un byte de la CPU en los diferentes modos de lectura y escritura. Contiene además los 4 latches a los cuales no se puede acceder desde el exterior; estos cogen en cada acceso de la CPU un byte de cada plano. En un acceso de lectura se envía un plano a la CPU (dependiendo del registro GDC 4), en un acceso de escritura se enlazan los latches con este byte (según el registro 3) y se vuelven a escribir a los planos. Se accede al GDC de la misma forma que al CRTC a través del puerto de índice (dirección *3ceh)* y un puerto de datos (dirección *3cfh).*

Registro GDC 0: Set / Reset		
Bit	**Significado**	**Acceso**
7-4	reservado	
3	Valor set / reset para el plano 3	RW
2	Valor set / reset para el plano 2	RW
1	Valor set / reset para el plano 1	RW
0	Valor set / reset para el plano 0	RW

Explicación:

Bit 3-0 Indican el valor de *set /reset* para cada plano, si está activada la función *set /reset* (ver registro 1).

Registro GDC 1: Enable Set / Reset (Activar set /reset)		
Bit	**Significado**	**Acceso**
7-4	reservado	
3	Función set / reset activada (1) para el plano 3	RW
2	Función set / reset activada (1) para el plano 2	RW
1	Función set / reset activada (1) para el plano 1	RW
0	Función set / reset activada (1) para el plano 0	RW

Explicación:

Bit 3-0 Un bit activado indica que la función *Set /Reset* está activada para este plano. En este caso no se enlaza el correspondiente latch con el byte de la CPU sino con *0* (bit correspondiente en el registro 0= 0) o *0ffh* (bit =1). Esta función no está disponible en el modo de escritura 1.

Registro GDC 2: Color compare (comparación de color)		
Bit	**Significado**	**Acceso**
7-4	reservado	
3	Valor de comparación de color para el plano 3	RW
2	Valor de comparación de color para el plano 2	RW
1	Valor de comparación de color para el plano 1	RW
0	Valor de comparación de color para el plano 0	RW

Explicación:

Bit 3-0 Estos bits juegan su papel en el modo de lectura 1, ver Registro 5, bit 3 (Modo lectura)

Registro GDC 3: Function Select (selección de función)		
Bit	**Significado**	**Acceso**
7-5	reservado	
4-3	Selección de función	RW
2-0	Contador de rotación	RW

Explicación:

Bit 4-3 Indican el tipo de enlace entre el byte de la CPU y los cuatro latches:

0: Move (sobrescribir)
1: AND
2: OR
3: XOR

Estos enlaces se ignoran en el modo de escritura 1, donde siempre se escribe directamente.

Bit 2-0 Indican en cuantos bit se rotará un byte de la CPU hacia la derecha antes de enlazarlo con los latches.

Registro GDC 4: Read Plane Select (selección del plano de lectura)		
Bit	**Significado**	**Acceso**
7-2	reservado	
1-0	selección de plano (plane select)	RW

Explicación:

Bit 1-0 Indican como Integer de 2 bit el plano al que accederá la CPU en un acceso de lectura.

Registro GDC 5: GDC Mode		
Bit	**Significado**	**Acceso**
7	reservado	
6	modo 256 colores	RW
5	Shift (desplazar)	RW
4	Modo Odd/Even (par / impar)	RW
3	Modo lectura	RW
2	reservado	
1-0	Modo escritura	RW

Explicación:

Bit 6 Activa el modo de 256 colores, que tiene una distribución de planos completamente distinta, por lo que debe ser activado de forma explícita.

Bit 5 Se coloca en el modo CGA 320 x 200, a fin de generar cuatro colores.

Bit 4 Activa el modo *Odd/Even (par/Impar)* desde el punto de vista del GDC (1). Deberá corresponderse siempre con el bit 2 del registro TS 4 (¡bit invertido!), ver allí la descripción.

Bit 3 Indica el modo de lectura activo:

Modo 0: El GDC lee los cuatro latches y manda el contenido a la CPU que ha sido seleccionada por el registro 4.

Modo 1: El GDC lee los 4 latches y compara sus bits individuales con el registro de comparación de color (*Color-Compare)* (si no ha sido excluido por el registro 7 - *Color Care)*. Se compara primero la combinación de los bits 0 de los cuatro latches, luego de los bits 1, etc. En cuanto coinciden completamente, se coloca el correspondiente bit en el byte enviado a la CPU.

Bit 1-0 Indica el modo de escritura actual:

Modo 0: El byte de la CPU se enlaza con los cuatro latches y según el registro TS 2 se escribe a determinados planos. Solamente se enlazan los bits que han sido autorizados por el registro 8. Si está activada la función *Set/Reset* (ver registro 1) para el plano, no se utilizará para el enlace el valor del byte de la CPU sino el valor *Set /Reset*. Este modo es muy apropiado para la manipulación de bits individuales.

Modo 1: En este modo el contenido de los latches se escribe directamente al plano seleccionado por el registro TS 2; el byte de la CPU carece de importancia. Este modo sirve para efectuar copias rápidas de grandes zonas de la pantalla.

Modo 2: Aquí se amplían los cuatro bits inferiores de la CPU hasta un byte (0 se convierte en 0 y 1 en 0ffh) y se enlazan con los correspondientes latches como en el modo de escritura 0.

Modo 3: Cada bit de la CPU que no ha sido desenmascarado mediante el registro 8 se enlaza con un bit del registro *Set / Reset*, y escrito al plano correspondiente a este bit. Así el bit 0 se enlaza con el bit 0 del registro *Set /Reset* según el registro de *selección de función* 3 y se escribe al plano 0 como bit 0, luego se enlaza el bit 0 con el bit 1 del registro *Set /Reset* y se escribe al plano 1. Lo mismo se ejecuta para todos los bits seleccionados del byte de la CPU.

Registro GDC 6: Miscellaneous (varios)		
Bit	**Significado**	**Acceso**
7-4	reservado	
3-2	Memory Map (mapa de memoria)	RW
1	Mode Odd / Even (Par /Impar)	RW
0	Modo gráfico	RW

Explicación:

Bit 3-2 Indica la zona en la cual se introduce la memoria de vídeo:

0: 0a0000h-0bffffh
1: 0a0000h-0affffh
2: 0b0000h-0b7fffh
3: 0b8000h-0bffffh

Bit 1 Un 1 indica que está activo el modo par / impar.

Bit 0 Si este Bit contiene un 1, estará activo un modo gráfico, en otro caso un modo de texto.

Registro GDC 7: Color Care		
Bit	**Significado**	**Acceso**
7-4	reservado	
3	Comparación de color activa para plano 3	RW
2	Comparación de color activa para plano 2	RW
1	Comparación de color activa para plano 1	RW
0	Comparación de color activa para plano 0	RW

Explicación:

Bit 3-0 Un bit activado incluye el correspondiente plano en la comparación de color en el modo de lectura 1 (ver registro 5); un 0 lo excluye.

Registro GDC 8: Bit Mask		
Bit	**Significado**	**Acceso**
7-0	Máscara para el byte de la CPU	RW

Explicación:

Bit 7-0 Un bit activado significa que el correspondiente bit de la CPU se enlaza con los latches; un bit desactivado significa que el contenido de los latches se mantiene sin modificación.

Attribute Controller (ATC)

La tarea de este controlador es la administración del color, es la penúltima instancia de la VGA. La utilización de este controlador es un poco más complicada que la de los demás: en los accesos de escritura solamente se tiene a disposición un puerto. En *3c0h* se encuentra el *Index/Data-Flip-Flop*, lo que significa que cada acceso de escritura a este puerto lo cambia entre el modo de índice y el modo de datos. Para que exista una situación de partida definida, el registro puede ser colocado explícitamente en el modo índice mediante un acceso de lectura al *Input Status register 1* (Puerto *3dah* en modo color, *3bah* en monocromo). Un acceso de escritura se efec-

túa después escribiendo primero el índica a *3c0h* y a continuación directamente el byte de datos al mismo puerto; después se puede escribir directamente el siguiente índice.

El acceso de escritura es un poco diferente: después de haber escrito el índice se puede leer el byte de datos a través del puerto *3c1h*; un acceso de lectura al puerto *3c0h* solamente devolvería el índice.

Una particularidad del registro Index/Data es la configuración del mismo en el puerto *3c0h*. Los bits 4-0 representan habitualmente el índice, el bit 5 por el contrario tiene otro significado:

Registro ATC : Puerto de índice /datos 3c0h (Index/data Port)		
Bit	**Significado**	**Acceso**
7-6	reservado	
5	Acceso a la RAM de la paleta	RW
4-0	Índice ATC	RW

Explicación:

Bit 5 Un valor 0 permite el acceso de la CPU a la RAM de la paleta (registro 0f), pero desconecta para ello el ATC, de forma que cambia la imagen al color del marco. Después de una modificación este Bit deberá ser colocado en cualquier caso en 1.

Bit 4-0 Indican el índice del registro ATC interno. Este valor se escribe a través del puerto *3c0h*, y se lee a través del *3c1h*.

Registro ATC 0f : RAM de paleta		
Bit	**Significado**	**Acceso**
7-0	Color DAC	RW

Explicación:

Bit 7 A cada valor de color EGA se le asigna a través de este registro el color real. En la EGA se colocan aquí directamente los componentes rojo, verde y azul, en la VGA se interpone el DAC *(Digital to Analog converter)*, convertidor analógico digital, el cual simula este orden en sus primeros 16 registros de la paleta. A través de este registro se pueden colocar por ejemplo los colores de texto a nuevos valores DAC.

Registro ATC 10h : Control de modo (Mode Control)		
Bit	**Significado**	**Acceso**
7	origen de las conducciones de color 4 y 5	RW
6	PelClock / 2	RW
5	Habilitar panning de píxeles (enable pixel panning)	RW
4	reservado	
3	Activar parpadeo	RW
2	habilitar gráficos línea (enable linear grahpics)	RW
1	Monocromo (1) / Color (0)	RW
0	Gráfico (1) / Texto (0)	RW

Explicación:

Bit 7 En los modos gráficos con 16 o menos colores este bit decide sobre el origen de las conducciones de color 4 y 5. Un 1 significa que estos bits provienen del registro 14 (Selección de color) Bits 0 y 1, un 0, en cambio, carga estas conducciones con los valores de los registros de la paleta. Si se utiliza el método de selección de color se podrá desplazar la zona DAC de la paleta en bloques de 16 unidades, sucediendo lo mismo en los modos EGA y CGA.

Bit 6 Un 1 reduce a la mitad la velocidad con la cual se envían los datos de píxeles al DAC, sistema que se utiliza en el modo 320 x 200 x 256, ya que en este solamente se utiliza la mitad de la resolución horizontal.

Bit 5 Un 1 impide el panning de píxeles debajo de la línea de división (Split-line), un 0 provoca un panning de toda la pantalla.

Bit 3 Un bit activado permite la libre utilización de todos los 16 colores de la paleta ATC, un 1 enciende el parpadeo. Tanto en el modo texto como en el modo gráfico, si este Bit está colocado se irá cambiando en el plano de intensidad entre la parte superior e inferior de la paleta, de forma que solo queden ocho colores disponibles. La estructura de la paleta es por lo demás libre, por lo que se pueden conseguir casi todos los efectos de parpadeo.

Bit 2 Si este bit contiene un 1, en los modos texto de 9 puntos (por ejemplo, modo 3) la octava línea se duplica cuando contiene caracteres con el código ASCII entre *0c0h y 0dfh*, a fin de enlazar caracteres de línea sin espacios. Si contiene un 0 la novena columna se borra, o bien se extrae del plano de intensidad (plano 3).

Bit 1 Un 1 activa los atributos monocromos (parpadeo, subrayado), un 0 los atributos de color.

Bit 0 Un 1 coloca el ATC en modo gráfico, un 0 en modo texto.

Registro ATC 11h: Overscan Color		
Bit	Significado	Acceso
7-0	Color DAC para el marco de la pantalla	RW

Explicación:

Bit 7-0 Determinan el número de color que se utilizará en las zonas *Overscan*. También aquí se utiliza en las tarjetas EGA un valor RGB de 6 bits, efectuándose la mezcla en la VGA mediante el DAC.

Registro ATC 12h : Habilitar plano de color (Color plane Enable)		
Bit	Significado	Acceso
7-6	reservado	
5-4	reservado, en muchos casos configuración de los bits de prueba	RW
3-0	Habilitar plano	

Explicación:

Bit 3-0 Activa (1) o desactiva (0) el correspondiente plano, eliminándose determinados colores (modelos de 16 colores) o puntos (modelos de 256 de colores) de la representación.

Registro ATC 13h : Panning de Píxeles Horizontal (Horizontal pixel panning)		
Bit	Significado	Acceso
7-4	reservado	
3-0	panning de píxeles horizontal	RW

Explicación:

Bit 3-0 Indican la cantidad de píxeles por la cual se desplazará la totalidad de la imagen (modo texto o modo gráfico) hacia la izquierda. Los valores están divididos de una forma un poco diferente: en los modos de 9 píxeles un valor de 8 indica que no se va a efectuar ningún desplazamiento, los valores 0-7 en cambio provocan un desplazamiento de 1-8 puntos. En los modos gráficos esto no tiene mucho sentido, ya que se podría trabajar de la misma forma utilizando la dirección de inicio, pero en el modo texto se puede conseguir un scrolling muy fino (ver apartado 6.6).

Registro ATC 14h : Selección de color (Color Select)		
Bit	**Significado**	**Acceso**
7-4	reservado	
3-2	Conducción de color 7-6	RW
1-0	Conducción de color 5-4	RW

Explicación:

Bit 3-2 Estos bits indican en los modos gráficos con menos de 256 colores los dos bits superiores de cada valor de color que se envía al DAC, reprogramando este registro los 16 colores EGA se pueden trasladar a los offsets de la paleta 0, 64, 128 y 192.

Bit 1-0 Si el Bit 7 del registro *Mode Control* está activado, estos dos bits representarán en los modos gráficos con menos de 256 colores los bits de color 5 y 4, pudiendo desplazarse de esta forma los colores EGA en combinación con los bits 3-2 a cualquier offset de color divisible entre 16.

Digital to Analog Converter (DAC)

El DAC es la última instancia en la generación de color, convirtiéndose aquí los colores de 8 bits provenientes o bien directamente de la RAM (modos de 256 colores) o generados por el bit 4 del ATC, mediante la paleta externa en señales analógicas, que son enviadas al monitor de forma separada en rojo, verde y azul. *Externo* significa que originariamente esta RAM estaba situada fuera de la placa principal, aunque hoy en día la técnica ha conseguido unificarla. EL DAC utiliza por lo tanto una paleta externa que contiene 256 valores de color, donde cada color está compuesto por 3 bytes, uno para rojo, verde y azul. utilizándose de estos solamente los 6 bits inferiores. De esta forma se puede seleccionar para cada color una tonalidad de 2 elevado a 18 = 262144 colores. Si se desea leer o situar una paleta, deberá indicarse primero el color inicial a partir del cual se desea leer o escribir, y a continuación se deberán enviar de forma seguida las órdenes de entrada/salida para los componentes rojo, verde y azul del color individual. Se pueden escribir todos los valores uno detrás de otro, ya que el DAC incrementa automáticamente el puntero. No es aconsejable utilizar el método "compatible" de control de la paleta a través del BIOS. Tampoco es mucho más compatible, y el tiempo de ejecución pude ser hasta diez veces mayor. El acceso a esta paleta lo permite el DAC a través de un total de cinco registros:

Máscara de píxel	RW
Puerto	3c6h

Este registro contiene normalmente el valor *0ffh*, representando una máscara mediante la cual se pueden recrear determinados colores sobre otros. Cada vez que el DAC desea leer un registro de la paleta durante la creación de la imagen, primero se combina el número de color con este registro.

Dirección de escritura del pixel	RW
Puerto	3c8h

Antes de un acceso de escritura hay que comunicar al DAC mediante este registro cuál es el color que se desea modificar, pudiéndose colocar después a través del puerto *3c9h* (valor de color del pixel) la paleta o parte de ella.

Dirección de lectura del pixel	WO
Puerto	3c7h

Antes de un acceso de lectura hay que comunicar al DAC mediante este registro cuál es el color que se desea leer, pudiéndose leer después a través del puerto *3c9h* (valor de color del pixel) el valor del color.

Valor del color del pixel	RW
Puerto	3c9h

En este puerto están colocados por el DAC los valores de los colores para su lectura o escritura, después de haberse iniciado el proceso de lectura o escritura a través de la *Dirección de escritura de pixel o Dirección de lectura de pixel.*

Estado del DAC	RO
Puerto	3c7h

Los dos bits inferiores de este registro indican el estado de acceso actual del DAC: un 0 significa que el DAC ofrece datos de lectura, un *11b*, en cambio, indica que el DAC está preparado para un acceso de escritura. Las tarjetas Super-VGA aún disponen de más registros, tanto registros individuales como registros de control ampliados. En parte hasta poseen completos controladores adicionales, que se encargan por ejemplo de la gestión de la memoria. Pero dado que estos registros no están normalizados de ninguna forma, no podemos efectuar aquí una descripción generalizada. Le recomiendo que se dedique a investigar, a ver si descubre algún efecto nuevo.

5. Modo X - el secreto mejor guardado

A pesar de que la utilización del modo *13h* parezca tan simple, tiene algunos inconvenientes bastante graves, que excluyen prácticamente la programación de buenos efectos especiales. A pesar de los grandes avances experimentados, los procesadores y la arquitectura actual de los buses no son capaces de calcular grandes áreas de la pantalla durante el retorno de un rayo.

5.1 El poder gráfico del modo X

Cuando no es posible calcular la estructura de la pantalla durante un retorno del rayo, la calidad de la imagen pierde bastante. Un ejemplo práctico: un sprite se mueve horizontalmente por la pantalla; si el rayo catódico sobrepasa esta zona durante la modificación de la memoria de vídeo, representará en la parte superior de la pantalla la nueva imagen, pero en la parte inferior la imagen anterior aún no modificada. En el mejor de los casos se producirá un parpadeo de la imagen, pero normalmente se consigue un sprite distorsionado y partido. Si durante la generación de la imagen no se pueden efectuar modificaciones, y durante el retorno del rayo tampoco se pueden efectuar modificaciones importantes, solamente nos queda una posibilidad: la imagen deberá ser calculada de forma casi invisible y deberá activarse durante el retrace vertical, por lo cual se necesitarán por lo menos dos páginas enteras de pantalla, una que se muestra actualmente mientras que la otra se está calculando.

Estas páginas pueden ser colocadas una detrás de otra en la memoria de vídeo y seleccionarse mediante los registros *0ch y 0dh* (dirección de inicio de línea). Pero en el modo *13h* el problema radica en que la imagen entera ya ocupa casi 64 Kbytes (exactamente *64000 bytes*). Una página de pantalla ocupa, por lo tanto, la totalidad de la memoria de vídeo insertada en la memoria principal (*0a0000h-0affffh*), lo cual imposibilita a la CPU acceder a la segunda página de pantalla. Las modificaciones solamente se podrán efectuar a través de los selectores de segmento de la VGA, los cuales se programan de forma diferente en cada fabricante. La VGA de IBM no ofrece ninguna posibilidad. La solución al problema se encuentra en el modo X, que vamos a explicar a continuación.

Otra importante característica negativa es la velocidad de acceso. Las imágenes en movimiento suelen tener siempre un fondo estático, que tiene que ser copiado con cada pantalla nueva a la página actual. En el modo *13h* se necesita, a pesar del acceso rápido de doubleword a través de un bus local VESA, un 10% más que en el modo X para copiar toda una página de pantalla. En otros sistemas de bus esta diferencia aún suele ser mayor. Si se utiliza además, por ejemplo, el acceso WORD,

a fin de soportar también los 286, la copia puede volverse desesperadamente lenta. En el modo X, en cambio, tal como explicaremos en el siguiente capítulo, se copian en un acceso a un byte cuatro píxeles de una sola vez, y además se utilizan los modos de lectura y escritura *1*, los cuales no necesitan ni complicadas transformaciones de direcciones ni envían datos a la CPU, lo cual conlleva la ventaja en velocidad frente a los accesos de 32 bits de la CPU.

5.2 Inicialización

¿Qué características debe tener el modo X? (La denominación "Mode X" se ha convertido en los últimos tiempos en una expresión casi oficial, y siempre hace referencia al mismo modelo de memoria. También admite resoluciones mayores, como *320 x 400*, en todas las VGA, hasta una resolución de *320 x 480* se puede conseguir en "casi" todas las VGA, pero, por desgracia, solamente "casi" en todas, y, por otro lado, una página de pantalla necesita aquí 128 Kbytes, de forma que se puede olvidar la ventaja de las diferentes páginas de pantalla. Lo más importante es la desconexión del mecanismo *Chain 4*, de forma que vuelve a ser posible el acceso libre a los diferentes planos, y además hay que asegurarse que el *modo par/impar* (Odd/Even; selección del plano a través del bit de offset inferior) esté desactivado, para lo que habrá que desactivar en el registro TS 4 (*memory mode*) el bit 3 (*enable chain 4*) y activar el bit 2 (*odd/even mode*):

```
port [$3c4]: = 4
port [$3c5]: = port [$3c5] )and (not 8) or 4;
```

Según la tarjeta gráfica (viva la compatibilidad) aún habrá que cambiar el acceso de memoria a un direccionamiento de bytes, o sea desactivar primero el direccionamiento *doubleword* del bit 6 del registro CRTC 14h (*Underline Row Adress*) y colocar el bit 6 del registro CRTC 17h (*CRTC Mode*):

```
port [$3d4]: = $14;
port [$3d5]: = port [$3d5] and (not 64);
port [$3d4]: = $17;
port [$3d5]: = port [$3d5] or 64;
```

Lo lógico ahora es borrar la memoria de vídeo, ya que en las partes no utilizadas por el modo *13h*, que ahora son visibles, aún puede quedar "basura" procedente de otros modos de vídeo. Para borrar se puede aprovechar el registro TS 2 (*Write Plane Mask Register*); en el cual para borrar la memoria de vídeo se activan todos los planos, de forma que 32000 accesos WORD o 16000 accesos DWord sean suficientes para borrar las cuatro páginas de pantalla.

Ahora ya tenemos ante nosotros una pantalla en modo X virginal, por lo que se nos deberá ocurrir algo para aprovecharlo, ya que ni una BGI ni el BIOS van a darnos soporte alguno, dejando de lado la posibilidad de abandonar el modo X y volver al modo texto. Para generar imágenes en el monitor en este modo, deberemos recordar su estructura.

5.3 Estructura

Tal como hemos dicho antes, en todos los modos gráficos basados en planos se ocultan 4 bytes detrás de cada dirección de memoria, uno por cada plano. Los 4 bytes están colocados casi uno encima de otro en una dirección, de ahí la expresión plane (plano).

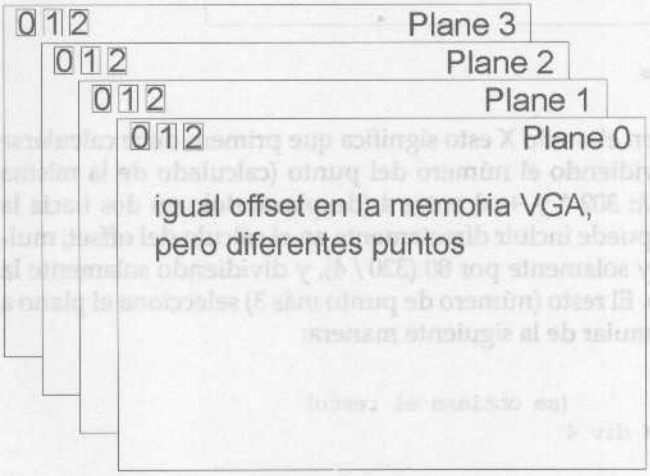

Figura 2: La representación de los planos

Estos planos son memorias casi independientes, a las que se puede acceder de forma individual, pero que durante la representación en pantalla se utilizan de forma paralela, lo que significa que los datos de los cuatro planos se leen simultáneamente.

En la estructura también existen diferencias pronunciadas entre los modos de 16 colores y el modo X. Dieciséis colores se pueden representar con 4 bits, por ello, los cuatro planos, ya que el bit 0 de un punto se guarda en el correspondiente bit del plano 0, el bit 1 en el plano 1, etc. En el modo X esto cambia debido a que los 4 bits ya no son suficientes para el direccionamiento de un punto, de forma que un punto precisa un byte de un determinado plano.

En esto los planos se rellenan a nivel de byte, lo que significa que el punto *0* se encuentra en el offset 0 del plano 0, el punto 1 en el mismo offset del plano 1, hasta el punto 4 que es el primero que está en el offset 1, volviendo a utilizar el plano 0.

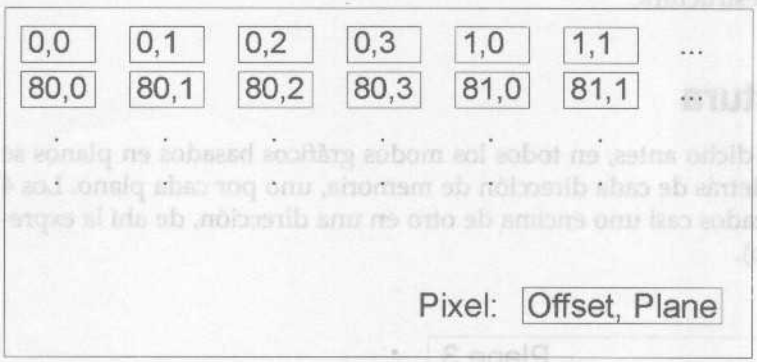

Figura 3: La estructura de los planos

Para el direccionamiento en el modo X esto significa que primero debe calcularse el offset de un punto, dividiendo el número del punto (calculado de la misma forma que en el modo *13h*: 302 * y + x) entre 4 (desplazándolo en dos hacia la derecha). Esta división se puede incluir directamente en el cálculo del offset, multiplicando la coordenada y solamente por 80 (320 / 4), y dividiendo solamente la coordenada x entre cuatro. El resto (número de punto más 3) selecciona el plano a utilizar. Esto se podría formular de la siguiente manera:

```
Plane:= X mod 4            (se obtiene el resto)
Offset:=  Y * 80 + X div 4
```

Este procedimiento es igual al que utiliza el modo *13h* de forma predeterminada, sólo que tiene una diferencia: el offset se consigue mediante un desplazamiento (shift) del número de punto en 2 bits hacia la derecha, y no mediante un enmascaramiento, de forma que en la memoria no se producen vacíos entre los puntos, entrando así cuatro páginas en la memoria de vídeo de 256 Kbytes.

Al seleccionar el plano en el modo X aún se deberá diferenciar entre accesos de lectura o de escritura: al leer, el número de plano se escribe al registro 4 (*Read Plane Select)* del GDC, al escribir, por el contrario, se puede acceder a varios planos al mismo tiempo (tal como hemos mostrado en el borrado de pantalla). Por ello, se coloca una máscara en el registro 2 (*Write Plane Mask)* del TS, que tiene que ser generada a partir del número de plano. En pseudocódigo, esto quedaría así:

```
Máscara:= 1 shl número_de_plano;
```

El plano 2 se convierte, por ejemplo, en la máscara *1 shl 2 = 4 = 0100*. Una palabra más sobre el direccionamiento a nivel de byte por parte de la CPU: siempre existe la tentación de acceder a los datos gráficos mediante accesos de 32 bits (¿para qué tenemos entonces un 386?) o utilizar por lo menos 16 bits (esto ya lo hacía en teoría el XT). Pero el que lo intenta aprenderá rápido: la imagen se presenta completamente distorsionada, lo que se debe a la forma de funcionar de la CPU durante los accesos a la memoria. Cuando la CPU, mediante *movsw*, copia un WORD (lo mismo es válido para un Doubleword) este proceso de copiado no se divide en movimientos de byte individuales, sino que el WORD se lee de forma completa y se escribe también entero.

Pero la VGA solamente puede admitir en sus latches, que sirven como memoria intermedia, 4 bytes. Por esta razón, después de un acceso de lectura, aquí solamente se encuentran los 4 bytes superiores (High-Bytes) del WORD leído. En el sigiuente acceso de escritura se determinará el correspondiente valor latch tanto en los cuatro bytes superiores como bajos. En la pantalla se ve que dos (en accesos de 32 bits hasta cuatro) bloques de cuatro tienen el mismo contenido, o sea que no se puede conseguir una imagen sensata.

5.4 Mayores resoluciones en el Modo X

El modo X tiene la gran ventaja de poder trabajar con cuatro páginas de pantalla, pero en la resolución no ha cambiado nada con respecto a su antecesor, el modo *13h*. Pero para determinadas aplicaciones hace falta una resolución mayor. Para ello existen las resoluciones SuperVGA, a las cuales se accede de forma diferente en cada tarjeta, problema que se puede solucionar mediante controladores VESA. El gran problema aparece cuando se desea utilizar la alta resolución junto con varias páginas de imagen. Aunque ya existan muy pocas tarjetas VGA que tengan menos de 1 Mbyte de memoria, de forma que en teoría se pueden guardar como mínimo dos páginas de pantalla en la memoria de vídeo en las resoluciones de 800x600, la administración de las páginas de pantalla de forma similar al registro *Linear Starting Adress*, no está prevista bajo VESA, por lo que una programación directa de las SuperVGA vuelve a provocar los inevitables problemas de compatibilidad.

En la búsqueda de un modo de mayor resolución que soporte el concepto de páginas, vemos que el modo X soporta hasta cuatro páginas enteras, aunque en muchos casos solamente se precisen dos, por lo que una duplicación de la resolución a 320 x 400 no es ningún problema. ¿Pero por qué 320x400 y no 640x200? Esto tiene que ver con la estructura en sí de los modos de 200 líneas de la tarjetas VGA. Las VGA son incapaces de acceder explícitamente a 200 líneas (en el *Miscellaneous-Output-Register*, bits 6-7, solamente están permitidos los valores 350, 400, 480 y en algunos casos 768 líneas de resolución vertical).

Pero que, a pesar de ello sean capaces, se debe a una posibilidad llamada *Double-Scan*, lo cual no significa otra cosa que en una resolución física vertical de 400 líneas cada línea se representa dos veces, duplicándose en dirección y, lo que produce una reducción de la resolución a la mitad. Para ello no hace falta modificar ni el timing vertical ni el horizontal, ya que físicamente se siguen representando 400 líneas con 320 puntos. Eso facilita enormemente la desconexión del mecanismo: en el registro 9 del CRTC (*Maximum Row Address*) deben borrarse 5 bits, el bit 7 y los bits 3-0. Según el BIOS de la VGA, la duplicación se efectuará a través del bit 7, previsto para ello (*Double Scan Enable*) o los bits 3-0, los cuales indican en el modo texto cuántas líneas de escaneo se precisan por línea de caracteres. Este valor tendrá que ser escrito al registro reducido en 1. En el modo gráfico, el valor del registro se corresponderá con la cantidad de copias que se realizan adicionalmente de la línea. Si aquí se utilizan valores mayores, la resolución vertical se podrá ir dividiendo hasta una resolución de 320 x 25, en conexión con el bit 7 hasta 320 x 12,5. Está claro que esto no tiene ningún sentido, pero nos explica el significado y la aplicación de este registro. Para volver a conseguir las 400 líneas originales, solamente habrá que borrar los bits del registro nombrados; en seudo-código:

```
CRTC [9] := CRTC [9] and 01110000b
```

Esta línea se encuentra en código ensamblador real en el procedimiento *Enter400* del módulo *ModeXLib.asm:*

```
Enter400 proc pascal far    ;pasa de modo X (200 líneas)
  mov dx,3d4h               ;al modo extendido de 400 líneas
  mov al,9                  ;CRTC, Registro 9 (Maximum Row Adress)
  out dx,al                 ;seleccionar
  inc dx                    ;leer valor
  in al,dx
  and al,01110000b          ;borrar bits 7 y 3:0
  out dx,al                 ;y escribir
  ret
Enter400 endp
```

Tal como hemos dicho, no hacen falta modificaciones en el timing, ni tampoco es necesario tener que reescribir las rutinas referentes al modo X, ya que la estructura se ha mantenido. En principio las 200 líneas del modo X normal al cambiar a 400 líneas se "comprimen" en el borde superior, y liberan la vista de las 200 líneas situadas por debajo, que son las que siguen directamente en la memoria de la VGA. Ahora se tiene a disposición para todas las rutinas un margen de valores para las coordenadas más amplio que puede ser utilizado libremente, ya que las preguntas de seguridad tampoco pintan nada aquí por razones de velocidad. El único cambio hace referencia al cambio de páginas de pantalla, ya que en vez de cuatro páginas de 64000 bytes se dispone ahora de 2 páginas a 128000 bytes.

Esto tiene su importancia al cambiar de página, ya que la dirección de inicio (*vpage)* ya no se cambia en este caso entre 0 y 16000, sino entre 0 y 128000. Además, ya no se dispone de una página "de reserva" para una imagen de fondo, ya que están ocupados casi la totalidad de los 256 Kbytes (256 Kbytes - 256000 bytes = 6 Kbytes que aún sobran al final del todo y que pueden ser utilizados para pequeños sprites de fondo). En este modo los fondos tienen que ser copiados desde la memoria principal a la VGA (mediante *p13_2_ModeX*, ver capítulo 5.6).

Al dibujar un fondo hay que tener en cuenta la extraña proporción de la página de 320:400, que genera unos rectángulos planos y anchos como puntos de imagen, de forma que los círculos dejan de ser círculos, y se convierten en elipses muy aplanadas. Cualquier dibujo deberá ser dibujado en una resolución "cuadrada" (por ejemplo, 640x480), y luego ser reducido a 320x400, ya que según nuestros conocimientos aún no existe ningún programa de dibujo que soporte una resolución de 320x400.

Aparte de esta nueva resolución también se pueden generar modos gráficos con mayor resolución, pudiéndose dar resoluciones bastante curiosas como, por ejemplo, 512x400 o 320x480. Además de la inicialización de estos modos, que es bastante más complicada, ya que precisa una reprogramación de los registros del timing, tampoco ofrecen ninguna ventaja con respecto a los modos SuperVGA, ya que sobrepasan con creces el gasto máximo de memoria de la programación por páginas.

En una VGA estándar una página se puede representar sin ningún tipo de problema, pero dos páginas ya sobrepasan los 256 Kbytes, que representan para los métodos de direccionamiento usuales la frontera superior de direccionamiento posible. Para direcciones superiores entra en juego la incompatibilidad de las tarjetas SuperVGA, de forma que es más conveniente utilizar directamente un modo SuperVGA, el cual, por lo menos, mantendrá una proporción de página correcta.

5.5 La utilización del modo X

En el modo X también se puede dar la necesidad de acceder a puntos de imagen individuales para modificar su color. En las aplicaciones que se basan casi exclusivamente en la construcción a base de puntos individuales (por ejemplo, desplazador de estrellas) es recomendable utilizar el modo *13h*, siempre y cuando se pueda prescindir de las cuatro páginas de pantalla. Este modo facilita bastante el direccionamiento de los puntos individuales, y además es más rápido, ya que la partición del offset en un offset de memoria y un número de plano se realiza de forma interna en la VGA, y no tiene que ser realizada por la CPU.

Por otro lado, se ahorra la indicación del número de plano a las direcciones de puerto del TS y del GDC, algo que puede ser bastante lento en algunas placas base

(algunos juegos de chips de ordenadores rápidos, a partir del 386, añaden a los accesos a puertos bastantes estados de espera, a fin de permitir al hardware la asimilación de los datos, algo que con el hardware disponible en la actualidad ya es superfluo).

Situar puntos

A fin de modificar algunos píxeles directamente, se puede utilizar el siguiente procedimiento, que también sirve para el desplazador de estrellas del modo *13h*, siempre y cuando se inicialice en vez del modo *13h* el modo X: LINIE_FK.PAS

```
Procedure PutPixel(x,y,col:word);assembler;
{coloca pto. (x/y) a color col (modo X)}
asm
  mov ax,0a000h          {cargar segmento}
  mov es,ax

  mov cx,x               {determinar Write Plane}
  and cx,3               {como x mov 4}
  mov ax,1
  shl ax,cl              {activar bit correspondiente}
  mov ah,al
  mov dx,03c4h           {Timing Sequencer}
  mov al,2               {Register 2 - Write Plane Mask}
  out dx,ax

  mov ax,80              {Offset = Y*80 + X div 4}
  mul y
  mov di,ax
  mov ax,x
  shr ax,2
  add di,ax              {cargar offset}
  mov al,byte ptr col    {cargar color}
  mov es:[di],al         {y colocar punto}
End;
```

Desenmascarando los dos bits inferiores de la coordenada X se determina primero el Plano, que tiene que ser reconvertido al formato del registro *Write-Plane-Mask*, colocando el bit que se corresponda con el número de plano. A continuación se calcula simplemente el Offset (desplazado en 2 bit hacia la derecha frente el modo *13h*), y se coloca el color.

Cambio de página

Sin lugar a dudas la mayor ventaja del modo X es la posibilidad de utilizar varias páginas de pantalla. En la mayoría de los casos estas páginas se utilizarán para ir cambiando constantemente entre las páginas de pantalla 0 y 1. Así, por ejemplo,

se dibuja la imagen nueva, con un sprite que ha avanzado, en la página que actualmente no está visible, se lleva esta a la pantalla, y se comienza de nuevo con las páginas invertidas.

En el modo de texto se tiene la función del BIOS 5 para cambiar de página, pero esto sirve de bien poco en el modo X. Pero a pesar de ello la cosa es bastante simple: como ya hemos dicho, las cuatro páginas de pantalla están situadas alineadas en la memoria: la página 0 ocupa el offset 0-15999, la página 1 el offset 16000-31999 etc., por lo que solamente se precisa la posibilidad de determinar la dirección en la cual comienza la representación de la página en el CRTC. Para ello sirven los registros del CRTC *0ch y 0dhk (Linear Starting Adresss,)*. El registro *och* contiene el High-Byte y el registro *0dh* el Low-Byte de la dirección en la cual comienza la imagen.

En general la dirección se puede establecer mediante el procedimiento *SetStart* en la *ModeXlib.asm*, al cual se transmite la dirección a colocar como parámetro. Este procedimiento divide la dirección recibida en High y Low Byte y la escribe al registro correspondiente.

El procedimiento *Switch* es un poco más especial, ya que utiliza la variable global *vpage* para guardar la dirección de inicio de la página que actualmente no es visible. Este procedimiento le ahorra de esta forma la administración de las páginas, de forma que puede crear siempre su imagen en la página establecida en *vpage*, que actualmente está invisible. Ejecutando *Switch* se visualiza esta página, y *vpage* marcará la nueva página invisible.

Los registros *0ch y 0dh* contienen, tal como hemos dicho, el offset del inicio de la pantalla, y no un número de página; de esta forma también se pueden dar valores intermedios, como veremos en el capítulo 6.3.

A fin de demostrar el principio de generación del juego de caracteres, tenemos el siguiente programa DRAW_FON.PAS, que no hace nada más que representar las introducciones a través del teclado utilizando el juego de caracteres PFONT4.GIF. Después de que el usuario finalice pulsando la tecla [ESC], aún se reproduce una cadena completa:

```
Uses Crt,ModeXLib,Gif,Font;

Var Entrada:Char;              {carácter introducido}
Begin
  Init_ModeX;                  {activar modo X}
  LoadGif('pfont4');           {cargar juego de caracteres}
  pl3_2_ModeX(48000,16000);  {y copiarlo a pág. 3}

  Repeat                       {bucle para visualizar entradas de teclado}
    Entrada:=ReadKey;          {obtener carácter}
    Print_Char(Entrada);       {y visual. en el monitor}
```

```
  Until Entrada=#27;          {hasta que se pulse ESC}

  Print_String('Hola, Test');{visualizar una cadena para terminar}
  ReadLn;
End.
```

El juego de caracteres presente como archivo GIF se carga primero en la página 3, que está "al final" de la memoria de vídeo, y que por lo tanto no molesta ni en animaciones de dos páginas ni a un scrolling (limitado) de la pantalla total.

En el bucle se incluye simplemente para cada introducción el correspondiente carácter, que se saca por pantalla. Al final aún se emite la cadena *Hola, Test*. Los dos procedimientos centrales *Print_Char y Print_String* se encuentran en la unidad *Font.pas:*

```
Unit Font;

Interface

  Procedure Print_Char(Chr:Char);
  {visualiza caracteres en modo X}
  Procedure Print_String(Str:String);
  {visualiza cadena en modo X}

  Procedure Scrl_Move;
  {mueve parte visible del Scrolly a la izquierda}
  Procedure Scrl_Append;
  {añade nuevos datos al scrolly por la derecha}

  Var Scrl_Y:Word;                {posición vertical del Scrolly}

  Const
    Scrl_Numero=4;
    {Nº de cadenas que hay en Scrl_Txt}
    Scrl_Txt:Array [1..Scrl_Numero] of String =
    {Sólo un texto de demo que se puede modificar de cualquier forma}
    ('Hola, esta es un Demo-Scroller del libro P C   A L     L Í M I T
E'
    +'  de MARCOMBO-Data Becker. Admitimos que no es precisamente el
más',
     'maravilloso, pero a cambio se conforma con un mínimo de esfuerzo'
    +'y sobre todo tiempo de cálculo. Incluso en ordenadores lentos ',
     'es posible emplear además otros efectos sin problemas; en cual-
quier caso,'
    +'el Scroller sólo necesita un 10 por ciento del tiempo de cálculo
',
     'disponible en un 486-40 con turbo apagado          Atención
Scrolly comienza '
    +'de nuevo
```

```
Implementation
Uses ModeXLib;
Const
  CharPos:Array[' '..'Z', 0..1] of Word=
  {posiciones y anchuras de los diferentes caracteres,
   bytes direccionados de CPU}
   ((71,4),(0,0),(0,0),(0,0),(0,0),(0,0),
    (0,0),(0,0),(0,0),(0,0),(0,0),(0,0),
    (1906,3),(1909,3),(1912,3),(1915,4),      {,-./}
    (3600,5),(3605,3),(3608,5),(3613,5),      {0..3}
    (3618,5),(3623,5),(3628,5),(3633,5),      {4..7}
    (3638,5),(3643,5),(3648,3),(3651,3),      {8..;}
    (3654,5),(3659,5),(3664,5),(3669,4),      {<..?}
    (0,0),(0,5),(5,5),(10,5),(15,6),(21,5),   {@..E}
     (26,4),(30,7),(37,5),(42,3),(45,4),(49,5), {F..K}
    (54,4),(58,8),(66,5),(1840,7),(1847,5),   {L..P}
    (1852,7),(1859,5),(1864,4),(1868,4),      {Q..T}
    (1872,5),(1877,6),(1883,8),(1891,5),      {U..X}
    (1896,5),(1901,5));                       {YZ}

Var Cur_X,                        {posiciones x e}
    Cur_Y:Integer;                {y actuales del cursor}

    Scrl_Number,                  {nº de la cadena scroll actual}
    Scrl_Pos,                     {posición en esta cadena}
    Scrl_ChrPos:Word;             {posición en el caracter}

Procedure Print_Char(Chr:Char);
{visualiza caracteres en pantalla en el modo X y mueve el cursor
 una posición}
Begin
  Chr:=UpCase(Chr);               {solo emplear mayúsculas}
  If Chr in [' '..'Z'] Then Begin {¿está en juego caracteres?, sí:}
    If 80- Cur_X <                {¿queda espacio?}
    CharPos[Chr,1] Then Begin
      Cur_X:=0;                   {no, siguiente línea, x a 0}
      Inc(Cur_Y,25);              {e y una altura de carácter más}
    End;
  Copy_Block(Cur_Y*80+Cur_X,48000+Charpos[Chr,0],CharPos[Chr,1], 22);
  {copiar caracteres a posición de cursor de posición de fuente
   (de tabla CharPos)(Cur_Y * 80 Bytes por línea + Cur_X) (altura
   22 líneas}
  Inc(Cur_X,CharPos[Chr,1]);      {mover cursor en anchura de carácter}
  End;
End;

Procedure Print_String(Str:String);
{visualiza una cadena en pantalla modo X,
 emplea para ello Print_Char}
Var i:Word;
```

```
Begin
  For i:=1 to Length(Str) do      {enviar cadena completa a Print_Char}
    Print_Char(Str[i]);
End;

Procedure Scrl_Move;
{simplemente desplaza el contenido de pantalla en el lugar del
Scrolly una posición a la izquierda, es decir, copiar 79 bytes de po-
sición x 1 a pos. x 0}
Begin
  Copy_Block(Scrl_y*80, Scrl_Y*80 +1, 79,22);
End;

Procedure Scrl_Append;
Var Chr:Char;                     {letra actual}
Begin
  Chr:=UpCase(Scrl_txt[Scrl_Number,Scrl_pos]);
                                  {obtener letra, sólo mayúsculas}
  If Chr in [' '..'Z'] Then Begin  {¿carácter está en el juego?, sí:}
    If CharPos[Chr,1] > 0 Then  {sólo representar caracteres existen-
te}
      Copy_Block(Scrl_y*80+79,48000+CharPos[Chr,0]+Scrl_ChrPos,1,22);
                                  {copiar 1 columna del juego de carac-
                                  {teres a borde izq. de pantalla}
    Inc(Scrl_ChrPos);             {y siguiente columna del caracter}
    If Scrl_ChrPos >= CharPos[Chr,1] Then Begin
      Inc(Scrl_Pos);              {si carácter listo, siguiente}
      Scrl_ChrPos:=0;             {y columna de nuevo a 0}
      If Scrl_Pos > Length(Scrl_Txt[Scrl_Number]) Then Begin
        Inc(Scrl_Number);         {si cadena lista, sig. cadena}
        Scrl_Pos:=1;              {Posición de nuevo a 0}
        If Scrl_Number > Scrl_Numero Then Begin
          Scrl_Number:=1;         {si texto acabado, al principio}
          Scrl_Pos:=1;
          Scrl_ChrPos:=0;
        End;
      End;
    End;
  End;
End;

Begin
  Cur_X:=0;                       {Cursor a esq. sup. izq.}
  Cur_Y:=0;

  Scrl_Y:=50;                     {valor por defecto para pos. y}
  Scrl_Number:=1;                 {inicio cadena 1, carác.1, columna 0}
  Scrl_Pos:=1;
  Scrl_ChrPos:=0;
End.
```

Lo primero que interesa es la lista de posición y anchura de los caracteres. Aquí se indica para cada carácter desde el espacio en blanco hasta la Z mayúscula su offset en la memoria de vídeo más su anchura en bytes. Todos los valores hacen referencia a bytes direccionables por la CPU, se accede, por lo tanto, a bloques de cuatro. Una offset de 71 para el espacio en blanco significa que el carácter comenzará en la línea 0 en la posición X 71 * 4 = 284; además tiene una anchura de 4*4=16 puntos.

Un anchura de *0* indica que se trata de un carácter no definido. En este caso, en el momento de la representación mediante copia, se copian *0* bytes, o sea que el carácter no se refleja. Por lo tanto, los caracteres desde la señal de exclamación hasta el + no están definidos, pero pueden ser "añadidos" en cualquier momento. Los procedimientos nombrados utilizan las dos variables *Cur_X y Cur_Y*, que contienen la posición actual del "cursor", o sea que indican, mediante las coordenadas x e y, en qué posición se representará el siguiente carácter.

Print_Char se encarga de la reproducción del carácter. Para ello se convierte primero en mayúsculas. Si se añaden a la fuente los caracteres en minúsculas, esta orden se podrá saltar, pero no le permitirá teclear solamente mayúsculas. Si el carácter convertido se encuentra en la tabla, primero se comprueba si la línea aún tiene espacio suficiente para representar el carácter; si no cabe, se salta a la siguiente línea. Los caracteres que no están en la tabla, por ejemplo los acentos (que pueden ser añadidos en cualquier momento por usted), tampoco se representan.

La orden *Copy_Block* sirve para transmitir los datos del carácter a la página 0. Como destino se indica, en este caso, *Cur_Y*80+Cur_X*, lo que se corresponde con el offset de la posición actual del cursor. La fuente es el offset extraído de la tabla, incrementado con el offset de la página del juego caracteres 3. La anchura también se extrae de la tabla, mientras que la altura se mantiene en un 22 constante. Una vez que ya se ha desplazado la posición del cursor en uno hacia la derecha, finaliza el procedimiento.

Print_String no hace nada más que activar la orden *Print_Char* para cada carácter de la cadena recibida, representándose por lo tanto este carácter. El resto de la unidad sirve para la realización del scrolly, que se describe a continuación. Pero antes vamos a dar el procedimiento que s encarga de la copia de los caracteres: *copy_block*, situado en *ModeXlib.asm*.

```
copy_block proc pascal far destino,fuente,breite,altura:word
local sprung:word
  mov dx,3ceh                    ;GDC
  mov ax,4105h                   ;ReadMode 0, WriteMode 1
  out dx,ax                      ;en Registro 5 : GDC Mode
  mov dx,3c4h                    ;TS
  mov ax,0f02h                   ;activar todos los planes
  out dx,ax                      ;en Registro 2 : Write Plane Mask
```

```
    push ds
    mov ax,0a000h                ;copiar en VGA
    mov es,ax                    ;-> ambos segmentos a 0a000h
    mov ds,ax
    mov si,fuente                ;datos fuente de la fuente
    mov di,destino               ;copiar a destino
    mov dx,altura                ;copiar líneas altas

    mov ax,80                    ;calcular salto entre dos líneas
    sub ax,breite                ;(= 80-ancho)
    mov sprung,ax
line_lp:
    mov cx,breite                ;cargar anchura
    rep movsb                    ;copiar una línea
    add si,sprung
    add di,sprung

    dec dx                       ;contador de líneas avanza
    jne line_lp

    pop ds
    ret
copy_block endp
```

Lo primero que hace este procedimiento es activar, mediante el registro GDC 5, el modo de lectura 0 y el modo de escritura 1, los cuales permiten una copia rápida de bloques de cuatro. Mediante el registro TS 2 se activan todos los planos.

A continuación se coloca *DS:SI* en la esquina superior izquierda del bloque que se desea copiar y *ES:DI* en las coordenadas destino. Ambas coordenadas se indican como Offset y solamente tienen que ser cargadas en los correspondientes registros. Como contador de líneas sirve *DX*, mientras que *CX* se carga después de cada línea con la anchura.

Después de cada línea copiada hay que saltar al inicio de la siguiente. Primero se calcula la distancia de salto: la distancia entre dos líneas es de 80 bytes, por lo que la distancia de salto será esta anchura de línea menos la anchura del bloque. El resto del procedimiento está compuesto por los dos bucles para los valores x e y (*rep movsb*). Pero lo que usted desea no es copiar líneas estáticas a la pantalla, sino que el objetivo de este capítulo es la creación de un scrolly. Para ello sirven los demás procedimientos y variables de la unidad *Font.pas*.

Dado que Pascal solamente puede controlar cadenas de una longitud de 255 caracteres, no queda otro remedio en el caso de textos largos que dividirlos en varias cadenas. La cantidad de cadenas parciales se indica en la constante *Scrl_Anzahl*. Los textos en sí se encuentran en la constante tipificada *Scrl_Txt*.

Como nuevas variables aparecen *Scrl_Number, Scrl_Pos y Scrl_ChrPos*, donde la primera contiene el número de la cadena parcial activa. La segunda variable contiene el número del carácter actual dentro de la cadena y la última la columna dentro de este carácter (la mayoría de caracteres son más anchos que una columna, con lo que deben ser representados en varios pasos).

El primer paso dentro de un bucle del scrolly es desplazar los caracteres ya visibles en 1 posición hacia la izquierda. Para ello se vuelve a utilizar la orden *Copy_Block*, que desplaza el contenido del scrolly de la posición x 1 a 0.

Scrl_Append añade finalmente los nuevos datos al borde derecho de la pantalla. Para ello se vuelve a leer el carácter actual (en mayúscula) del array y se comprueba si existe en la tabla. Se puede dar que determinados caracteres no estén definidos, de forma que en el siguiente paso se tienen que controlar las anchuras de carácter 0, ya que en este caso no se descartan automáticamente.

Los caracteres definidos disponibles se copian ahora mediante *Copy_Block* al borde derecho de la pantalla (*Scrl_y*80 +79*, o sea coordenada x 79), teniéndose en cuenta al mismo tiempo la columna actual dentro del carácter (*Scrl_ChrPos*). Ahora se incrementa la columna en 1. Si el carácter ya es visible completamente, el contador de columnas se restablece a 0 y el contador de caracteres avanza. Cuando este llega al final de la cadena se restablece a 0 y se incrementa el contador de cadenas. Cuando este llega al final, se vuelve a comenzar con el principio del texto, restableciéndose los valores de todas las variables a sus valores originales. Como programa de demostración sirve el archivo SCROLLY.PAS

```
Uses Crt,Tools,ModeXLib,Gif,Font;

Var Sinus:Array[0..127] of Word;{tabla seno para oscilación vertical}
    t:Word;                     {"Tiempo", posición en el seno}

Begin
  Init_ModeX;                   {activar modo X}
  LoadGif('pfont4');            {cargar juego de caracteres}

  p13_2_ModeX(48000,16000);     {y copiar a pág. 3}
  Sin_Gen(Sinus,128,Scrl_y div 2,Scrl_y div 2);
                                {preparar tabla seno para mov. vert.}
  t:=0;                         {tiempo comienza en 0}

  Repeat
    WaitRetrace;                {sincronización}
    Scrl_Move;                  {mover parte visible a la dcha.}
    Scrl_Append;                {colocar nueva columna la izq.}
    SetStart(Sinus[t and 127]*80);  {obtener mov. vertical}
```

```
    Inc(t);                    {seguir en tabla seno}
  Until KeyPressed;
  TextMode(3);
End.
```

Primero se carga el juego de caracteres que se utilizará para representar los caracteres del scrolly. En el bucle principal se espera al retrace, se desplazan los caracteres visibles en una columna (4 puntos) hacia la izquierda y se añaden nuevos datos mediante *Scrl_Append*.

Además, se incluye aquí un movimiento vertical, desplazándose el comienzo de la pantalla mediante una tabla senoidal hacia arriba o hacia abajo; para controlar este procedimiento sirve la variable *t*, que indica la posición actual dentro de la tabla senoidal.

5.6 Ampliación del cargador GIF al Modo X

Como ya hemos visto, el formato GIF es el formato gráfico más apropiado para demos. En el modo X nos hará falta, por lo tanto, una posibilidad de cargar y visualizar imágenes GIF. Esto no plantea ningún problema, ya que el cargador está diseñado de tal manera que desempaqueta la imagen en la memoria principal. En el modo*13h* se copia desde allí simplemente con una orden de Pascal *Move* hacia la memoria de vídeo. El último paso, la copia, deberá escribirse nuevamente, a fin de adecuarse a la diferente organización de la memoria del modo X.

Para ello sirve el procedimiento *p13_2_ModeX* en *ModeXLib.asm*. Este procedimiento copia una imagen del tamaño *pic_size*4* desde la memoria principal (puntero en *Vscreen)* al modo X a partir de la dirección de inicio *start*. Para copiar, por ejemplo, una imagen de 320 x 200 puntos cargada mediante *LoadGIF* a la página de pantalla 2, la orden sería:

```
P13_2_modex(2*16000,64000 div 4);
```

El problema de un procedimiento de copia de este tipo consiste en dividir la imagen entre las cuatro planos diferentes; el procedimiento más simple, pero al mismo tiempo el más lento, sería actuar punto a punto, o sea cargar un punto origen, seleccionar el plano destino y volver a escribir el byte. Una forma mucho más elegante es copiar primero todos los puntos del plano *0* (se encuentran en los offset *0, 4, 8*... de la imagen origen), luego del plano 1, etc. , ya que de esta forma se evita el cambio constante del registro del timing.

```
p13_2_modex proc pascal start,pic_size:word
    mov b plane_1,1        ;guardar máscara plano
    mov w plane_pos,0
    push ds
    lds si,dword ptr cs:vscreen  ;cargar dirección origen
```

```
        mov w plane_pos,si      ;y guardarla
        mov ax,0a000h           ;establecer direcc destino
        mov es,ax
        mov di,start
        mov cx,pic_size         ;obtener número
    @lpplane:
        mov al,02h              ;TS Register 2 (Write Plane Mask)
        mov ah,b plane_1        ;enmascarar plane adecuado
        mov dx,3c4h
        out dx,ax

    @lp1:
        movsb                   ;copiar byte
        add si,3                ;posicionar en el siguiente byte origen
        loop @lp1
        mov di,start            ;obtener de nuevo posición destino
        inc w plane_pos         ;dirección fuente a nuevo inicio
        mov si,w plane_pos
        mov cx,pic_size         ;obtener tamaño
        shl b plane_1,1         ;enmascarar siguiente plane
        cmp b plane_1,10h       ;¿copiados los 4 planes?
        jne @lpplane

        pop ds
        ret
    Endp
```

La variable *plane_1* que contiene la máscara de plano actual, se inicializa primero con *1*, a fin de acceder al plano *0*. El puntero a los datos origen se carga en *DS:SI* y su offset se guarda en *plane_pos*. Después de cargar la dirección destino en *ES:DI* y la longitud de imagen en *CX*, comienza el bucle de planos *@lpplane*, en el cual se selecciona el plano correcto mediante la máscara actual (Timing Sequenzer Registro 2, *Write Plane Mask*). El bucle interno *@lp1* copia cada vez un byte y salta 3 bytes en la imagen origen (*movsb* ya ha incrementado *SI* en *1*), a fin de poder leer el siguiente byte del mismo plano.

Después de ejecutarse el bucle, un plano ya se encontrará en su posición correcta. Para copiar el siguiente plano, se sitúa el puntero destino al principio, a continuación se incrementa la dirección destino en 1, a fin de leer, por ejemplo, al copiar el plano *1* los offset *1, 5, 9* etc. *CX* se vuelve a cargar y se enmascara el nuevo plano mediante una rotación hacia la izquierda en 1. La condición de interrupción es el enmascaramiento de un quinto plano no existente, o sea en cuanto se han copiado los cuatro planos.

De esta forma se encontrará en la pantalla del modo X una página de imagen de la memoria principal, pasada en el formato del modo *13h*. Desde allí se puede copiar mediante rápidos modos de copiado a cualquier página que se desee, por ejemplo para borrar una imagen antigua y restaurar el fondo anterior.

El método utilizado de copiar a nivel planos también se utiliza para la representación de sprites, tal como describimos en el capítulo 7. Pero dado que allí se aplican unas determinadas condiciones especiales, se utiliza un procedimiento completamente nuevo.

5.7 Un simple scrolly de texto

Aquel que ya haya trabajado alguna vez con un C-64, ST o similar, aún los debe conocer en su forma original: los scrollys. Letras en movimiento, en parte con contenidos muy interesantes, cuya lectura era casi obligada. Hoy en día el contenido suele tener un papel secundario, ya que la presentación se realiza de una forma mucho más cara y efectista. Pero los principios básicos continúan siendo iguales, pudiendo ser explicados de la mejor forma mediante el ejemplo de un "scrolly clásico".

Un scrolly en el modo X

En el fondo un scrolly no tiene que hacer más que desplazar un texto de izquierda a derecha por la pantalla, "presentándolo" por lo tanto al ojo del observador. En el modo texto esto se puede programar en pocos minutos: se coge una cadena de caracteres, un puntero que muestra la posición del borde izquierdo de la pantalla dentro de la cadena, y se escriben por ejemplo 70 veces por segundo 80 caracteres partiendo de la posición actual dentro de una línea de la pantalla y se incrementa el puntero en una posición. De esta forma se desplaza la ventana, o sea la parte actualmente visible del texto, en la dirección de lectura por encima de la cadena, y ya tenemos nuestro scrolly.

En el modo texto esto es bastante más complicado: el principio es el mismo, pero el programador solamente dispone (en el modo *13h*) de pocas rutinas (del BIOS), o de ninguna (en el modo X), para la representación de caracteres, de forma que uno se tiene que fabricar todo por su cuenta, algo que en el fondo ya sería apropiado a fin de conseguir una velocidad de representación razonable.

Una gran diferencia frente al scrolly de texto es también el desplazamiento del contenido de la pantalla. En el modo de texto bastaba con reproducir con cada renovación de la pantalla la completa cadena de caracteres en su nueva posición. Pero en el modo gráfico el factor velocidad tiene mucha más importancia, de forma que es imposible permitirse el lujo de componer cada carácter a partir de los datos de la fuente.

Existe una solución mucho más simple: dado que el texto que está en pantalla (en un scrolly simple) ya no cambia, sino que simplemente se desplaza, lo que se puede hacer es simplemente copiarlo gráficamente a una nueva posición, desplazan-

do la columna 4-319 en 4 puntos hacia la izquierda y añadiendo una nueva parte a las columnas liberadas 316-319. Los pasos de una anchura de 4 puntos permiten la utilización efectiva de las ventajas del modo X, ya que aquí se pueden copiar de forma simple en el modo de escritura 1 bloques de una anchura de 4 byte.

Si se deseara efectuar un desplazamiento por un valor no divisible entre cuatro (por ejemplo 3), se deberían copiar los contenidos de plano a plano, algo que es solamente posible a nivel de puntos individuales. Pero una copia de offset a offset sin un cambio de plano permite la copia de cuatro puntos a la vez, lo que mejora ostensiblemente la velocidad.

Juego de caracteres para el texto en movimiento

Lo único que falta ahora es rellenar en el borde derecho de la pantalla una zona de 4 puntos de anchura con nuevas partes del texto, pero ¿de dónde se sacan los datos que se desean representar, o sea el juego de caracteres? Una posibilidad muy simple sería utilizar el juego de caracteres de la ROM de la VGA, pero esta fuente no ofrece ni color ni un diseño gráfico del otro mundo. Diseñar su propio juego de caracteres no es muy fácil, pero es algo básico para generar una presentación efectiva.

Muchos programadores se esfuerzan en escribir para ello su propio editor que contenga todas las posibilidades de un programa de dibujo y que pueda generar las fuentes más extraordinarias. Pero el esfuerzo que cuesta el desarrollo de un editor de este tipo no está en proporción con su utilidad. ¿Qué sentido tendría dedicar semanas o meses al diseño de un editor, cuando se tarda la mitad de tiempo en crear la Demo que utilizará la fuente? Además, no tiene mucho sentido que cada programador vuelva a inventar la rueda y programe nuevamente las funciones gráficas básicas - líneas, círculos, copiar partes, etc. -, cuando todos los programas de dibujo de dominio público ya dominan todas estas características a la perfección.

La solución con más sentido es utilizar alguno de estos programas de dibujo. Incluyendo un alfabeto completo en una única imagen se ahorra mucho espacio en el disco duro, ya que la imagen se guarda lógicamente en el formato GIF, el cual está diseñado para ahorrar espacio.

La forma de realizar los caracteres depende de cada uno. Los artistas no tardarán mucho en crear en poco tiempo una fuente completa, pero es más práctico utilizar primero una fuente ya incluida, y desarrollar a partir de esta la suya propia. Lo importante es que los caracteres sean válidos para su posterior utilización. Dado que la rutina que añade las nuevas columnas se base en los bloques de cuatro, habrá que diseñar los caracteres en base a estos bloques de cuatro. Para ello la coordenada X del carácter así como su anchura deben ser divisibles entre cuatro,

aunque no causa ningún problema que tengan una anchura diferente, por lo que se pueden utilizar fuentes proporcionales.

En cuanto se han ordenado todos los caracteres se llega a la parte cansada, la determinación de todas las posiciones de los caracteres. Gracias a estas posiciones el procedimiento para representar más adelante los caracteres tendrá acceso al juego de caracteres, leyendo en una tabla su posición en la memoria de vídeo, así como su anchura.

Para crear la tabla se coge papel, lápiz y un programa de dibujo que muestre en cada momento la posición actual del cursor. Hay que tener cuidado, ya que algunos programas de dibujo comienzan a contar las coordenadas en la posición 1, por lo que la esquina superior izquierda tiene la coordenada 1/1. Esto debe parecer más ergonómico a los desarrolladores de los programas de dibujo, pero no se corresponde con la realidad matemática ni con el funcionamiento de nuestro ordenador. Las coordenadas deben basarse en cualquier caso en un sistema basado en el 0, en el peor de los casos se deberá restar un 1 al valor que se obtenga (incluso en el posicionamiento de bloques de cuatro). Mediante las coordenadas del punto superior izquierdo se puede calcular el posterior offset en la memoria de vídeo mediante la fórmula ya conocida:

```
Offset := Y * 80 div 4
```

Hay que apuntar en la tabla este valor para cada letra. El plano es en cualquier caso el 0, lo cual era la condición para poder utilizar el rápido modo de escritura. Para determinar el ancho se mide primero en puntos (de cabeza o mediante las herramientas de medición del programa de dibujo), y se redondea hacia arriba hasta el siguiente múltiplo de 4, a fin de obtener la anchura en bloques de cuatro, que es elemental para este procedimiento, tal como hemos explicado. Este valor también debe ser apuntado en la tabla.

En cuanto se haya conseguido esto, habrá que trasladar la tabla al programa, manteniéndose en vistas a la simplicidad el orden ASCII. Así se deberá construir por ejemplo una tabla desde el código ASCII 32 (espacio en blanco) hasta el 96 (Acento "`"), y se incluye para los caracteres no existentes una anchura de 0 (imposible). Si se considera necesario también se podrán añadir las letras en minúscula (97-122) o todo el juego de caracteres ASCII, ya que se puede ocupar cada carácter con un pequeño gráfico. Pero todo esto es bastante cansado y no suele tener mucho sentido: normalmente se tiene bastante con las mayúsculas, los números y algunos caracteres especiales. En nuestro CD se incluye lógicamente una pequeña fuente de ejemplo, la cual es utilizada por el programa impreso anteriormente.

6. División de pantalla y otros efectos

¿Qué sería la programación gráfica sin efectos especiales? Se podrían mirar bonitas imágenes, hasta se podrían dibujar. Pero hasta la más pequeña animación o sprite en movimiento ya es un efecto gráfico. Pero no vamos tratar en este capítulo esta forma de efectos, que precisan, sobre todo, trabajo de la CPU, sino aquellos que pueden ser creados de forma independiente por la VGA, en los cuales la CPU simplemente da una orden.

6.1 Lo básico

Todos los efectos tratados se pueden crear de una forma u otra mediante el poder de la CPU, pero para ello se deberían mover ingentes cantidades de datos dentro de la memoria de vídeo. Los desplazamientos en memoria suelen ser por sí bastante lentos, solamente se pueden acelerar ligeramente mediante el DMA (en los ordenadores modernos el acceso directo de la CPU suele ser más rápido), y suelen fracasar en los accesos a la memoria de vídeo como más tarde cuando se llega al cuello de botella que es el bus de datos.

En el viejo bus ISA conseguir la velocidad de los datos no era posible de ninguna forma, ya que hasta en los ordenadores más potentes se trabajaba a 8 MHz, a fin de garantizar la compatibilidad hacia "abajo". Además de la tarjeta VGA de 16 bits se rebajaba a 8 bits cuando existía adicionalmente una tarjeta Hercules, de forma que los accesos aún tardaban el doble.

Hoy en día los modernos sistemas de bus trabajan de forma continuada con buses de 16 o hasta 32 bits y a un mínimo de 33 MHz, pero aún significan una "ralentización" para los datos, ya que se tienen que cumplir unos protocolos muy estrictos, se deben añadir cantidades increíbles de estados de espera (Waitstates) y algunas VGA ni llegan a absorber todo ello e incluyen sus propias esperas.

Por lo tanto debe existir algún otro camino para generar efectos especiales en la pantalla, cosa que se ve en las innumerables demos, las cuales, a pesar de los mejores movimientos y distorsiones, aún son capaces de calcular objetos en tres dimensiones y reproducir sonido. Para todo ello se utilizan lógicamente las posibilidades de las tarjetas gráficas.

A pesar de que las tarjetas de los PC (aún) no disponen de controladores programables, procesadores etc., concebidos expresamente para crear efectos gráficos, como por ejemplo posee el Amiga, y que reservan a la CPU la simple tarea de control, la gran cantidad de registros y las incalculables posibilidades de combina-

ción de sus contenidos ofrecen muchas posibilidades que antes los desarrolladores de la VGA ni se atrevían a soñar.

En esto se debería limitar el trabajo a los registros compatibles con la VGA original, ya que el traslado de las manipulaciones de registros específicos de una marca a otros juegos de chips sería muy difícil, cuando no imposible, por no disponer la nueva plataforma de un registro comparable.

Cuando se estén buscando nuevas aplicaciones de los registros se puede probar, basándose en una completa descripción de los registros, a adivinar qué pueden provocar las modificaciones, pero no se puede evitar experimentar con diferentes contenidos de los registros.

Siempre que no se toquen los registros del timing del CRTC, estos experimentos no pueden dañar o destrozar nada, apareciendo en el peor de los casos una imagen confusa, distorsionada o hasta ninguna imagen, pero pulsando la tecla [ESC] se vuelve al editor de Turbo Pascal, y todo arreglado.

El peligro aparece cuando se modifican los registros del CRTC 0-7, que hacen referencia sobre todo al timing horizontal, y los cuales en caso de mala utilización pueden "cerrar" el monitor o la tarjeta gráfica. Si se lía el timing, puede suceder que el rayo catódico se vuelva totalmente confuso y salte locamente por encima de la pantalla, o que utilice frecuencias demasiado altas, lo que puede producir un sobrecalentamiento de las bobinas del monitor, y de esta forma la destrucción de monitores baratos.

Por esta razón estos registros están protegidos a través del bit 7 del registro CRTC *11h*, el cual tiene que ser desactivado antes de poder efectuar manipulaciones de aquellos registros. Cuando se ha cometido un fallo y se está arriesgando la integridad del monitor, esto se suele notar en fuertes silbidos y parpadeos del monitor. Cuando se experimente con estos registros conviene, por ello, tener la mano en el enchufe de la corriente, a fin de poder desconectar el monitor en caso necesario.

En general debería evitarse completamente el trabajo con estos registros, a fin de eludir cualquier riesgo. Además hasta ahora no se han conseguido efectos de ningún valor reprogramando el timing, lo máximo que se ha logrado son imágenes distorsionadas. Pero todo ello no impide que experimentemos con los demás registros, los cuales, tal como explicaremos a continuación, ofrecen algunas posibilidades que seguro no estaban previstas en su forma original.

6.2 Así funciona: División de pantalla

Muchas veces se nos plantea la situación de tener que presentar dos zonas independientes en la pantalla, por ejemplo una parte con letra en movimiento (scroll), en la parte superior, y una línea de estado fija en la parte inferior. Esta tarea se puede traspasar lógicamente a la CPU. Pero si a uno le importa la velocidad de ejecución, seguro que buscará alguna posibilidad de programar el hardware, encargándole el trabajo a la VGA.

El secreto se encuentra en los registros *Line Compare* o *Split-Screen (división de pantalla)*. Aunque no ofrezcan la misma funcionalidad que los registros comparables de los ordenadores domésticos (Amiga, por ejemplo), tienen la suficiente capacidad para la mayoría de las aplicaciones. Este registro determina la línea en la cual se divide la pantalla en una parte superior movible y en una parte inferior estática. En los modos de 200 líneas hay que duplicar este valor.

El funcionamiento de este registro es muy simple: durante la representación de la imagen la VGA va contando permanentemente la línea actual. Para ello utiliza el número de línea físico, lo que quiere decir que también en los modos de 200 líneas como los modos *13h* o X este valor se mueve (a causa de la duplicación de líneas) entre *0 y 400*. Este número se encuentra en un registro al cual (por desgracia) no se puede acceder a través de direcciones de puerto, pero que se puede utilizar internamente para una comparación con el registro de *Line Compare* (CRTC, índice 18h). En cuanto alcanza el valor que indica este registro, el contador de direcciones se carga con un 0, lo que significa que la representación comienza en esta línea con los datos que se encuentran en el Offset 0 de la memoria de vídeo.

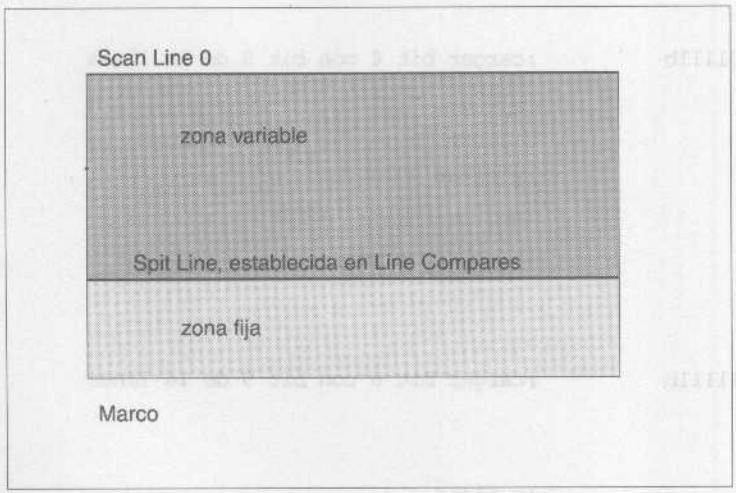

Figura 4: Estructura del Split-Screen

Dado que el número de línea no se puede guardar en 8 bits (la VGA no reconoce ningún modo con menos de 350 líneas, hasta los modos de 200 líneas son generados por 400 líneas físicas), el registro de *Line Compare* suele estar dividido entre tres registros, en las tarjetas Super-VGA hasta cuatro. Ya que existen más registros que sobrepasan los 8 bits, sobre todo en el caso del los registros del timing vertical, y por razones de espacio, no se añaden registros nuevos, sino que todos los bits "que desbordan" estos registros se resumen en dos nuevos registros (en la Super-VGA tres).

El primero de estos registros es el registro *Overflow* , el cual -nomen est omen- acoge los desbordamientos del timing vertical. Entre otros se encuentra aquí en el bit 4 (por otro lado es el único bit de este registro que no está protegido por el bit de protección del registro *11h*) el bit 8 del registro *Line Compare*. El bit 9 de este registro de comparación de línea se encuentra en el segundo registro de desbordamiento, el registro *Maximum Row Address*, concretamente en el bit 6. Si se desea, por lo tanto, colocar el *Line Compare*, los bits del número de línea deberán dividirse mediante desplazamientos de bit y enmascaramientos entre estos 3 registros. Esta tarea es realizada por el procedimiento *Split*, que se encuentra en *ModeXlib.asm*.

```
Split proc pascal far row:byte  ;Screen-Splitting en columna row
  mov bl,row
  xor bh,bh
  shl bx,1                      ;*2 por duplicación de líneas
  mov cx,bx

  mov dx,3d4h                   ;CRTC
  mov al,07h                    ;Registro 7 (Overflow low)
  out dx,al
  inc dx
  in al,dx
  and al,11101111b              ;cargar bit 4 con bit 8 de la línea
  shr cx,4
  and cl,16
  or al,cl
  out dx,al                     ;y fijar

  dec dx
  mov al,09h                    ;Registro 9 (Maximum Row Address)
  out dx,al
  inc dx
  in al,dx
  and al,10111111b              ;cargar bit 6 con bit 9 de la línea
  shr bl,3
  and bl,64
  or al,bl
  out dx,al                     ;y fijar
```

```
    dec dx
    mov al,18h              ;Registro 18h (Line Compare/Split Screen)
    mov ah,row              ;activar los restantes 8 bits
    shl ah,1
    out dx,ax
    ret
  Endp
```

Lo primero que hace este procedimiento es doblar el valor recibido, a fin de poder utilizar en los modos de 200 líneas a pesar del *double-scan* (duplicación de líneas) las coordenadas -y reales entre 0 y 199. Después de seleccionar el registro *Overflow* (CRTC, registro 7), se borra primero el bit 4, a fin de poder colocar aquí el bit 8 del número de línea -mediante enmascaramiento y desplazamiento a la derecha del bit 4. De la misma forma se copia el bit 9 del número de línea al bit 6 del registro de *Maximun Row Address* (CRTC, registro 9). Finalmente se copian los restantes 8 bit del número de línea al registro *18h* (*Line Compare o Split Screen)* y se abandona el procedimiento.

Si se coloca el inicio de pantalla mediante el registro *Linear Starting Address 0ch y 0dh* en una posición mayor, por ejemplo el inicio de la página 2, se representará en la parte superior esta zona, y en la parte inferior debajo de la línea situada en el registro de comparación de línea (*Line Compare)* se mostrará el contenido del inicio de la memoria de vídeo. Ahora la parte superior se puede desplazar de la forma que se desee mediante la *Linear Starting Address*, y efectuar un scroll en cualquier dirección (vea el siguiente capítulo), mientras que la parte inferior mantiene siempre el mismo contenido. Esto es una solución un poco limitada frente a las posibilidades de los ordenadores de "juegos", pero es suficiente para crear algunos efectos interesantes. Aparte de la posibilidad de dividir la pantalla en dos partes claramente diferenciadas, también se puede pensar en efectos mucho más amplios: ¿qué pasaría por ejemplo si se anulara la división entre la parte movible y la parte estática? Dado que la línea de partición se puede determinar libremente, esto también se podría ejecutar dentro de un bucle. Como en esta línea comenzará siempre el offset 0 de la memoria, toda la parte inferior de la pantalla también se moverá cuando se modifique el registro de *Line Compare.* Una reducción en 2 (*¡Double Scan!)* desplazaría todo el contenido de la parte inferior de la pantalla en una línea hacia arriba, de forma que en el borde inferior aparecerá una nueva línea.

De esta forma se puede desplazar un imagen desde abajo por encima de otra imagen de fondo, pasando de esta forma, por ejemplo, a un nuevo capítulo de la demo. Para ello se carga simplemente en la página 1 la imagen de fondo (mediante *Load_Gif y P13_2_ModeX)* y la imagen "superior" a la página 0, a continuación se activa la división de pantalla, primero en la línea 400, de forma que la imagen superior esté invisible, y solamente se vea la imagen de fondo. Si ahora se desplaza la *Split-Line* hacia arriba (reduciendo el registro *Line Compare* en 2), se moverá el contenido de la página 0 desde abajo por encima de la imagen de fondo, hasta que

alcance el borde superior (*Line Compare 0*). Ahora se puede cambiar sin ningún tipo de problema a la página 0 y desactivar la división de pantalla -pero siempre en este orden, ya que, si no, se puede producir una corta aparición del fondo antiguo, debido a que la desconexión del *Split-Screen* trae la página actual a la pantalla, que en este momento ya debería ser la página 0.

Lógicamente este proceso también se puede hacer al revés, o sea tirar una imagen hacia abajo fuera de la pantalla, de manera similar a un juego de cartas, en la que el movimiento de una carta permite ver la carta situada por debajo, un efecto que es bastante bonito para una presentación (*Slideshow* es una representación programada de imágenes predefinidas en un intervalo de tiempo determinado para efectos de presentaciones, por ejemplo en un escaparate).

Para ello hay que volver a colocar la imagen del primer plano en la página 0 y la imagen de fondo en la página 1. Primero se activa la división de pantalla, se cambia a la página 1, y se incrementa en un bucle el valor del registro de *Line Compare*, de forma que la *Split-Line* se mueva hacia abajo. Por encima de esta línea irá apareciendo el contenido de la página 1, que es la que está activa en este momento.

El incremento o decremento del registro de *Line Compare* se debería efectuar, como en la mayoría de los efectos, de forma sincronizada con el retrace vertical, a fin de evitar saltos y centelleos. Para ello simplemente hay que añadir dentro del bucle la orden *WaitRetrace*, lo que sirve al mismo tiempo para ralentizar un poco el proceso, algo muy importante, ya que la CPU descargada de trabajo efectuaría esta tarea en fracciones de segundo. Mediante la sincronización ya sólo se presentarán 70 imágenes por segundo (frecuencia de repetición del modo *13h* y del modo X), de forma que el proceso ya durará por lo menos 2,8 segundos (200 líneas / 70 líneas por segundo). Al experimentar con estos efectos nos daremos cuenta enseguida que los colores se presentan de forma errónea, lo que se debe a la limitación a 256 colores de los modos gráficos basados en paletas. Al cargar una imagen se carga también su paleta, pero al cargar la segunda imagen se machaca la paleta anterior, ya que solamente se pueden representar 256 colores al mismo tiempo. La única solución a este dilema es la utilización de *una* sola paleta para ambas imágenes. En el caso de imágenes diseñadas por uno mismo se puede tener en cuenta durante el diseño en utilizar la misma paleta (normalmente se puede cargar y guardar de forma separada). Pero cuando se utilizan imágenes ya diseñadas, de colecciones o escaneadas, habrá que unificar ambas paletas. Muchos de los programas de conversión gráfica poseen esta función, pero en la mayoría de los casos se efectúa con poco calidad, lo que produce distorsiones de los colores y la aparición de puntos colocados erróneamente.

La mejor solución es reducir ambas imágenes de forma separada a 128 colores. Si se utiliza una imagen compleja y una simple, lógicamente se podrá efectuar una división de por ejemplo 192 y 64 colores, siendo lo único importante que la suma dé 256. A partir de esto ya no tendremos ningún problema en crear una paleta

completa mediante un convertidor gráfico. Ya sólo hará falta cargar las imágenes en la demo, dando igual el orden en el que efectuemos esta carga, ya que las paletas son iguales.

En este efecto también es posible, y tiene bastante sentido, contar el registro *Line Compare* en pasos de 1, de forma que las líneas de escaneo individuales, o sea la mitad de una línea gráfica, representen la unidad más pequeña, y el desplazamiento dure el doble. Aquí influye el movimiento, ya que la vista es incapaz en cualquier caso de reconocer las medias líneas. En una división de pantalla simple puede ser negativo que aparezca media línea gráfica en medio de una imagen, de forma que aquí se deberían utilizar números impares (la representación de la segunda imagen parcial comienza en la línea que sigue inmediatamente a la línea indicada en el registro *Line Compare*. Ya que esta tiene que tener un valor par, el valor del registro deberá ser impar).

Vamos a dar un pequeño programa de ejemplo para explicar la utilización del procedimiento de división. Este programa no hace nada más que desplazar dos páginas de un solo color una encima de otra, y luego las vuelve a separar. Para ello se llenan primero las dos páginas de pantalla y se activa la página 1 (dirección de inicio *16000*). En los bucles se va incrementando o decrementando el registro de comparación de línea, lo cual produce el movimiento deseado, y todo el proceso se sincroniza con el retrace. La introducción a través del teclado seguro que volverá locos a la mayoría de informáticos, ya que utiliza con el *Exit* una orden que no pega nada en el concepto de Pascal de programación estructurada. Pero ¿por qué nos vamos a complicar la vida (y hacerla más lenta) utilizando complicadas variables Flag como condiciones de interrupción de los bucles?

```
Uses Crt,Gif,ModeXLib;
Var i:Word;
begin
  Init_ModeX;
  LoadGif('uwp');
  p13_2_ModeX(16000,16000);
  LoadGif('enter320');
  p13_2_modex(0,16000);
  SetStart(16000);
  Repeat
    For i:=200 downto 0 do Begin
      WaitRetrace;
      Split(i);
      If KeyPressed Then Exit;
    End;
    For i:=0 to 200 do Begin
      WaitRetrace;
      Split(i);
      If KeyPressed Then Exit;
```

```
    End;
  Until KeyPressed;
End.
```

Por lo demás la *Split-Screen* funciona en todos los modos gráficos y de texto, pero sólo tiene sentido cuando se puede representar más de una página de pantalla. En el modo *13h*, por ejemplo, no existe suficiente memoria para controlar dos zonas de pantalla independientes, de forma que siempre se presenta el mismo contenido, solamente que se divide de manera diferente en las distintas ventanas.

Por esta razón lo explicado anteriormente hace referencia al modo X, aunque sea posible, como hemos dicho, crear una división de pantalla en los modos gráficos de una página; lo que ocurre es que esto carece de sentido.

6.3 Scrolling en las cuatro direcciones

Hasta ahora hemos utilizado el registro *Linear Starting Address* del CRTC (registros *0ch y 0dh*) para cambiar entre las páginas de pantalla. Para ello se calculaba el offset de la página (número de página * 16000) y se escribía a este registro. Pero de la misma forma se puede escribir valores intermedios, y desplazar de esta forma el principio de la zona representada de forma libre dentro de la RAM.

Cuando se construye una imagen nueva mediante el rayo catódico, este ya no recibe los datos de la imagen del principio de la memoria de vídeo, sino desde el principio del punto establecido por el valor del registro *Linear Starting Address*. De esta forma se puede desplazar la pantalla física, o sea la parte que en definitiva aparece en pantalla, como si fuera una ventana por dentro de toda la RAM de la VGA, de forma que cada vez se muestre un trozo de la imagen de 256 Kbyte.

Si se desplaza el inicio de la imagen dentro de la RAM, la ventana se desplazará en esta dirección, de forma que el gráfico del monitor se moverá en la dirección opuesta. Un incremento de la *Linear Starting Address* en 80 significa un desplazamiento del contenido de la pantalla en una línea (320 píxeles / 4 planos = 80 byte de longitud) hacia arriba.

El contenido de la línea 0 se encuentra fuera de la ventana, y la línea 200 aparece por abajo como última línea. Ahora estarán visibles las líneas 1-200. Incrementando el registro en pasos de 80 byte se puede conseguir un scrolling vertical de todo el contenido de la pantalla utilizando un mínimo de cálculo.

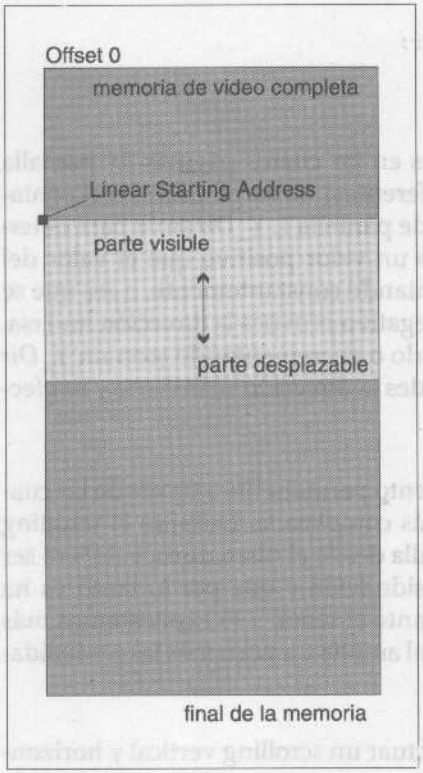

Figura 5: Estructura de la memoria de vídeo con cuatro páginas verticales

El siguiente programa de ejemplo realiza la función explicada anteriormente:

```
User Crt,Gif,ModeXLib;
Var y,
    y_dir:word;
Begin
    Init_ModeX;
    LoadGif('uwp');
    p13_2_ModeX(0,16000);
    p13_2_ModeX(32000,16000)
    LoadGif('enter320')
    p13_2_ModeX(16000,16000)
    p13_2_ModeX(48000,16000)
    y:=80
    y_dir:=80;
    Repeat
      Inc(y,y_dir);
      WaitRetrace;
      SetStart(y);
```

```
        if (y >= 600*80)
        or (y <= 80) Then y_dir:=y_dir;
      Until KeyPressed;
  End.
```

Después de cargar ambas imágenes parciales en las cuatro páginas de pantalla (lógicamente se pueden utilizar imágenes diferentes), se cargo primero el contador Y con 80, la dirección de inicio de la línea de pantalla 1. *Y_Dir* sirve para determinar la dirección del scrolling, significando un valor positivo que el valor del registro *Linear Starting Address* se irá incrementando constantemente, o sea que se ejecutará un scrolling hacia arriba, un valor negativo provoca la dirección inversa. Dentro del bucle principal se va incrementando o decrementando (con un *Y_Dir* negativo) la dirección de inicio (*y*). En los bordes la dirección se invierte y se efectúa un scrolling hacia atrás.

Cuando se desea generar un texto en movimiento permanente a través de las cuatro páginas de pantalla, la cosa se vuelve más complicada. Durante el scrolling habrá que cargar la siguiente página de pantalla desde el disco duro, y deberá ser copiada a la parte de la pantalla que ya ha sido leída y que por lo tanto ya ha desaparecido. Pero esto nos complicaría bastante el tema, y lo explicaremos más adelante en el efecto llamado *Efecto Pyro*, el cual amplía un poco más las posibilidades del scrolling.

Muchas veces se da la situación de querer efectuar un scrolling vertical y horizontal al mismo tiempo, por ejemplo para logotipos que ocupen cuatro páginas de pantalla y que solamente son visibles parcialmente, o sea que parece que se vayan moviendo por debajo de la pantalla.

La solución parece bastante simple: hasta ahora el registro *Linear Starting Address* solamente se movía en pasos de 80, ¿por qué no hacerlo en pasos de 1? En este caso, lo que se quita en el extremo izquierdo aparecería directamente en el borde derecho. Esto resulta porque las líneas siguen estando situadas una detrás de otra en la memoria.

Si se ha desplazado por ejemplo el inicio de la imagen en un byte (desplazamiento hacia la izquierda de 40 píxeles), el CRTC presentará en la primera línea los bytes 1 hasta 80 (en vez de los normales 0-79); pero el byte 80 ya pertenece en el fondo a la siguiente línea de la muestra, de forma que cada byte que se extrae del borde izquierdo aparece en el borde derecho y en una línea más arriba.

La solución a este problema radica en aumentar la pantalla de forma virtual a una anchura de 640 píxeles. Se seguirán mostrando 320 puntos, pero a la derecha "al lado" del monitor se añaden otros puntos teóricos, de forma que las cuatro páginas de imagen ya no están situadas una encima de otra como en el scrolling vertical, sino que forman un cuadrado. Los puntos que vayan desapareciendo por la

izquierda aparecerán en el borde extremo derecho (coordenadas x 636-639), en la zona invisible, y se traspasarán nuevos puntos desde la zona invisible a la zona visible. Dicho de otra forma, se puede desplazar la ventana, lo que es en el fondo la pantalla en el modo X, de forma horizontal, ya que ahora existe espacio para ello (las páginas de imagen situadas de forma horizontal una al lado de la otra).

¿Cómo se genera este modo? Como era de esperar, la VGA posee un registro para ello. Se trata en este caso del registro *13h* del CRTC: el offset de fila (*Row Offset*). Detrás de este nombre tan insignificante se esconden posibilidades insospechadas. Aquí se indica la distancia de salto mediante la cual se desplazará el puntero interno a los datos (*Linear Counter*) al llegar el rayo catódico al borde derecho de la pantalla. Esto es por lo tanto la separación de las líneas dentro de la memoria de vídeo, o sea su longitud!

Este registro contiene normalmente el valor 40, tanto en el modo *13h* como en el modo X, lo que equivale a una anchura de 80 byte. El registro cuenta en pasos WORD, o sea que una anchura de 80 byte se cuenta como 40 WORD. Aunque en el modo *13h* las línea tengan una longitud de 320 byte, la base de cálculo que se corresponde con la unidad máxima direccionable por la CPU se cuadruplica desde un byte hasta un DoubleWord, de forma que como valor programado vuelve a resultar 320 / 8 = 40.

También se pueden utilizar valores más altos. Si se escribe por ejemplo un 80 en este registro, esto significará que las líneas tendrán una separación, o sea una longitud, de 160 bytes (que se corresponde a 640 píxeles). Haciendo esto se producirán vacíos de una longitud de 80 bytes entre las diferentes líneas de 80 bytes, que serán rellenados por las mitades de líneas invisibles que sobran por el lado derecho.

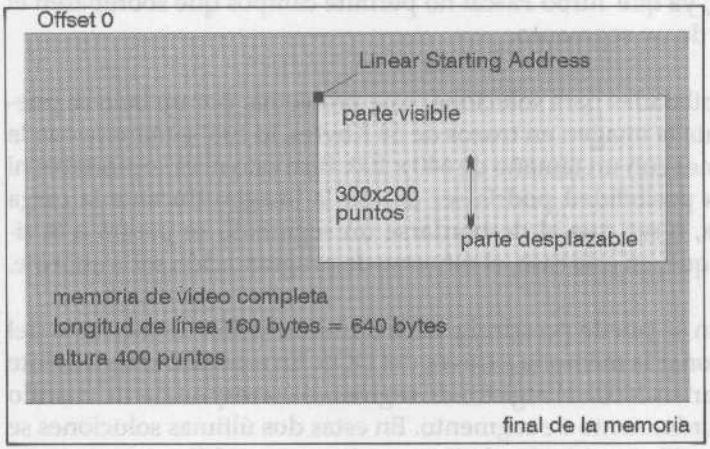

Figura 6: La ventana visible dentro de la memoria de vídeo

La figura anterior muestra la estructura de la memoria de vídeo después de activar el modo de 160 byte. Para encender esta duplicación de líneas sirve el procedimiento *Double* del módulo *ModeXlib:*

```
double proc pascal far
  mov dx, 3d4h
  mov ax, 5013h
  out dx, ax
  ret
double endp
```

Aquí no se realiza nada más que cargar el registro *13h* del CRTC con el valor *50h= 80d*, para duplicar la longitud de línea tal como hemos dicho a 160.

Utilización del cargador GIF para imágenes grandes

También es posible cargar imágenes GIF en este modo ampliado. Para ello se puede utilizar en principio el cargador normal, ya que trabaja de forma completamente independiente al formato de pantalla, lo que quiere decir que desempaqueta primero los datos comprimidos en el orden que los ha leído hacia la memoria principal, desde donde se efectúa la copia a la memoria de vídeo. Para cargar por ejemplo una imagen en el modo de 640 puntos, esta solamente deberá existir como una imagen con un anchura de 640 puntos, de otra forma se desplazará cada segunda línea a la parte derecha de la pantalla, lo cual no es nuestra intención.

De esta forma también se pueden cargar imágenes muy altas de una sola vez, por ejemplo para los créditos, generando y cargando por ejemplo una imagen de 320 x 800. Aquel que calcule el tamaño de la imagen llegará al conclusión de estas imágenes tienen un tamaño superior a 64 Kbytes, de forma que no pueden ser colocadas en la variable *Vscreen*, ya que Turbo Pascal no permite campos que sobrepasen el tamaño de 64 Kbytes de un segmento.

Existen diversas posibilidades para solucionar este problema: por un lado se puede dividir directamente la imagen en trozos de 64 Kbytes, lo cual generaría con la anchura doble imágenes con un tamaño de 640 x 100. Pero esto ni es confortable ni elegante. La segunda posibilidad podría ser dividir la imagen durante la carga entre varios punteros, o sea que al desbordarse un segmento se pasará a la siguiente variable. Aunque, en este caso, el esfuerzo de programación sería enorme.

Como tercera solución se puede prescindir de la administración de memoria del Turbo Pascal y direccionar la memoria a través del DOS. En este caso simplemente se debería incrementar en *1000h* el registro de segmento correspondiente cuando se produzca un desbordamiento de segmento. En estas dos últimas soluciones se nos presenta el mismo dilema: ¿de dónde sacamos la memoria? Para una imagen de dos páginas ya se necesitan 128 Kbytes, que en muchos casos ya serían demasiados, mientras que los 64 Kbytes se mantienen dentro de los límites soportables.

La solución que consideramos la más apropiada se encuentra en el procedimiento *ReadGIF* del *ModeXlib*. Aquí se comprueba directamente después de las dos órdenes *stosb* que escriben el punto descifrado si existe un desbordamiento. Si se ha producido, se copia la imagen calculada hasta ese momento a la memoria de vídeo. Dado que las páginas de pantalla grandes solamente tienen sentido en el modo X, se llama para ello al *p13_2_ModeX*, que traspasa la *Vscreen* a la memoria de vídeo a la posición definida por *VRAM_pos*, restableciendo luego el puntero a 0. La variable *VRAM_pos* se desplaza a continuación al final de la zona copiada dentro de la memoria de vídeo, de forma que un eventual nuevo desbordamiento se añada aquí.

Al final del procedimiento *ReadGIF* se carga la variable *Rest* con el contenido actual del *Vscreen* (se encuentra en *DI* e indica directamente el número de bytes copiados, ya que Pascal coloca las variables siempre en direcciones de segmento enteras), para que en el programa principal mediante otra llamada al *p13_2_ModeX* y con este resto como cantidad de bytes por copiar se copie el resto a la memoria de vídeo.

Aunque este procedimiento no sea el más elaborado algoritmicamente, tiene dos ventajas fundamentales: por un lado el procedimiento *ReadGIF* es 100% compatible con el procedimiento antiguo, lo que quiere decir que no hay que decidir sobre el tamaño de la imagen ni mediante traspaso de parámetros ni mediante lectura de la cabecera GIF. De forma que, en teoría, se acepta cualquier formato de imagen (tampoco representa ningún problema la ampliación a resoluciones Super-VGA), y por otro lado se ahorra una gran cantidad de memoria, ya que se sigue ocupando solamente un segmento en la memoria principal, que puede ser borrado después de la carga por el heap y utilizarse para la carga de la siguiente imagen.

En cuanto se ha cargado una imagen de este tamaño en la memoria de vídeo, comienza el scrolling en sí. Para ello se vuelve a utilizar el registro *Linear Starting Address*, pero que ahora se puede modificar con cualquier valor: de esta forma se puede conseguir un scrolling vertical, horizontal y hasta diagonal. Para el scrolling vertical se actúa de la forma descrita en el capítulo anterior: el registro *Linear Starting Address* se va incrementando o decrementando en una longitud de línea. Para ello solamente hay que tener en cuenta la longitud de línea modificada, de forma que se deben utilizar pasos de 160 bytes para el scrolling vertical. El scrolling horizontal se consigue incrementando o reduciendo el valor del registro en pasos de 1, representando en bytes cuatro píxeles, de forma que el scrolling solo es posible en pasos de 4 píxeles (otra solución se muestra en la descripción del registro *Pixel-Panning 13h* del controlador de atributos *Attribute controller*).

El scrolling en diagonal se compone de estas dos direcciones. Para efectuar por ejemplo un scrolling en 4 puntos hacia arriba a la izquierda se incrementa la *Linear Starting Address* en 641 (160 bytes por línea * 4 líneas + 1 byte). Las capacidades de este registro quedan demostradas con el siguiente programa:

```
Uses Crt,Gif,ModeXLib;
Var x,
    x_dir,
    y,
    y_dir:word;
Begin
  Init_ModeX;
  double;
  LoadGif('enter640');
  p13_2_ModeX(vram_pos,rest div 4);
  x:=1;
  x_dir:=1
  y:=160
  y_dir:=160
  Repeat
    Inc(x,x_dir);
    Inc(y,y_dir);
    WaitRetrace;
    SetStart(y+x);
    if (x >= 80)
    or (x <= 1) Then x_dir:=-x_dir;
    if (y >= 200*160)
    or (y <= 160) Then y_dir =-y_dir;
  Until KeyPressed;
End.
```

En esta versión del scrolling se necesitan, además de las variables ya conocidas para la posición y dirección Y, las respectivas variables para la dirección X. Estas se llaman *x y x_dir* y cumplen exactamente la misma misión que sus semejantes en dirección Y: *x* indica la posición actual en dirección X, donde hay que tener en cuenta que un incremento o reducción en 1 se corresponde con un desplazamiento de cuatro puntos, debido a los cuatro planos paralelos, *x_dir* describe la dirección del scrolling X, significando un 1 por paso por el bucle un desplazamiento del contenido de la imagen en 4 píxeles hacia la izquierda, un -1 un desplazamiento en 4 puntos hacia la derecha.

Después de la inicialización ya conocida se activa en esta rutina mediante la llamada a *Double* el modo de 160 bytes de la VGA, para duplicar de forma virtual la anchura de la pantalla, y permitir de esta forma un scrolling horizontal. Seguidamente se carga la imagen mediante *LoadGIF*, debiéndose tener en cuenta que a causa del tamaño de la imagen (640 x 400 píxeles = 256000 bytes) ya se copiará durante la carga una parte hacia la memoria de vídeo (más bien se descarga por falta de memoria).

Pero dado que esta descarga solamente se produce cuando ocurre un desbordamiento por encima de la frontera de 64 Kbytes, se traspasa el resto de la imagen a la memoria de vídeo mediante una llamada al procedimiento *p13_2_ModeX*, en concreto a partir de la posición que sigue al último byte copiado (*vram_pos*) con la

longitud *Rest*, que *LoadGIF* carga en esta variable. Aún se deberá dividir el número de bytes entre cuatro, ya que el *p13_2_ModeX* trabaja con bytes CPU, o sea que en una longitud de 1 y a causa de los cuatro planos se copian 4 bytes, indicándose el contenido de la variable *Rest* como bytes reales, o sea correspondientes al número de píxeles.

La dirección X se inicializa tal como hemos dicho con 1, de forma que primero se efectúa un scrolling hacia la izquierda; la posición también se sitúa en 1, para comenzar con la columna 1 (antes de entrar en el bucle la pantalla aún comienza en la columna y fila 0). En las variables de la dirección Y resalta la duplicación de los valores a 160, a causa de la longitud de línea duplicada a 160 bytes.

En el bucle en sí cambia bastante poco: para efectuar un movimiento en dirección X se incluye una nueva orden *Inc*. Por lo demás se carga el inicio de imagen con la suma de los offset *x* e *y*, de forma que se tengan en cuenta ambas direcciones (ver arriba). A parte hay que añadir otra condición que invierta la dirección X cuando se llegue al borde izquierdo o derecho, de forma que junto a una condición similar para la dirección Y se mantenga la zona del scrolling en el marco de la pantalla virtual dada anteriormente por la memoria de vídeo.

6.4 Todo a la vez - División de pantalla en combinación con el scrolling

La división de pantalla y un scrolling multidireccional pueden crear efectos atractivos por su cuenta, pero en combinación su efecto óptico aumenta considerablemente, tal como muestra el programa *Scrl_spt.pas:*

```
uses crt,Gif,ModeXLib;
Var x,                      {offset actual en direcc. x}
    x_dir,                  {indica direcc. de scroll para x}
    y,                      {offset actual en direcc. y}
    y_dir:word;             {indica direcc. de scroll para y}
    split_line:word;        {posición actual de la Split-Line}
    split_dir:word;         {indica direcc. de mov. de la Split-Line}
Begin
  Init_ModeX;               {activar modo X}
  double;                   {activar modo 160 bytes}
  Screen_Off;               {apagar pantalla}
  LoadGif_Pos('640400',160*50);  {cargar imagen grande en (0/50)}
  p13_2_ModeX(vram_pos,rest div 4);   {copiar resto a memoria VGA}
  LoadGif('corner');        {cargar imagen pequeña en (0/0)}
  p13_2_ModeX(0,160*50);    {y copiar a pantalla}
  Screen_On;                {activar pantalla}

  split_line:=150;          {Split en linea 150}
```

```
    split_dir:=1;                {Split-Line hacia abajo}
    x:=1;                        {comienzo x en col. 1}
    x_dir:=1;                    {direcc. x 1 byte por pasada}
    y:=160;                      {comienzo y en lín. 1}
    y_dir:=160;                  {direcc. y +160 bytes por pasada}
    Repeat
      Inc(x,x_dir);              {movimiento x}
      Inc(y,y_dir);              {movimiento y}
      Inc(Split_line,Split_dir);   {mover Split Line}
      WaitRetrace;               {esperar retrazado}
      SetStart(50*160+y+x);      {y escribir nuevo inicio en registro,}
                                 {saltar las 1ªs 50 líneas}
      Split(Split_line);         {partir pantalla en Split Line}
      if (x >= 80)               {borde x alcanzado -> invertir direcc. x}
      or (x <= 1)                Then x_dir:=-x_dir;
      if (y >= 200*160)          {borde y alcanzado -> invertir direcc. y}
      or (y <= 160)              Then y_dir:=-y_dir;
      if (split_line >= 200) {alcanzado borde split -> cambiar direc-
ción}
      or (split_line <= 150) then split_dir:=-split_dir
    Until KeyPressed;            {hasta tecla}
  End.
```

En este programa no se divide como hasta ahora la pantalla en una línea determinada en dos zonas estáticas, sino que se incluye esta separación dentro del movimiento. Para describir este movimiento sirven las variables *Split_Line y Split_dir*, que utilizan el mismo principio que las correspondientes variables *x e y*: *Split_line* contiene la línea actual en la que se efectúa la división, y *Split_dir* tiene o bien el contenido 1 para un movimiento positivo en dirección Y (hacia abajo) o un -1 para un movimiento de la línea de división hacia arriba.

Después de activar el modo X y las líneas de 160 bytes se apaga la pantalla por razones de mejora óptica durante la carga. La ventaja de velocidad casi no se nota, pero es más bonito que el monitor se quede en negro durante la construcción de la imagen. Si se desea ver el proceso de carga (a efectos de control , se pueden anular estas ordenes, pero en una versión definitiva de la demo no debe ser visible la construcción de la imagen durante la carga.

La imagen grande, que es la que se desplazará más adelante en todas las direcciones, se carga con la ayuda del procedimiento *LoadGIF_Pos* en la posición 160*50, o sea 50 líneas después del inicio de la RAM, mientras que la imagen pequeña se coloca mediante un *LoadGIF* normal al principio de la memoria. El *LoadGIF_Pos* trabaja de la misma forma que el *LoadGIF*, pero recibe además un parámetro adicional para el offset al cual se copiará el desbordamiento en caso de imágenes grandes.

Gracias al desplazamiento en 50 líneas se cumplen las condiciones para la división de la pantalla: dado que el *Linear Counter* se sitúa a 0 en la línea de división (Split-

Line), se representará debajo de esta línea el contenido del principio de la memoria; aquí está situada la imagen pequeña. Para el scrolling se utiliza más adelante la dirección de inicio de la imagen grande como origen. A continuación se vuelve a activar la imagen, y la línea de división se coloca en el valor inicial 150. Este valor se convierte internamente en líneas de pantalla, o sea en líneas gráficas, de forma que gracias a la división el cuarto inferior de la pantalla recibe su propia autonomía. *Split_dir* tiene el contenido 1, de forma que en el bucle se desplaza la parte inferior de la pantalla hacia abajo. Si la línea de división tiene el valor 200, lo que correspondería a una "desaparición" completa de la parte inferior de la división, se invierte la dirección del movimiento; lo mismo sucede en el borde superior (150), de forma que además de la imagen superior en movimiento se mueve la inferior de forma vertical.

La orden *SetStart*, que se encarga del scrolling, se ha ampliado en esta versión con la constante 50*160, que se encarga que las primeras 50 líneas de la memoria de vídeo no lleguen nunca a la pantalla durante el scrolling, ya que contienen la parte inferior de la división de pantalla, y la imagen grande comienza 50 líneas más abajo.

6.5 Cerrar una puerta - juntar una imagen

Otra posible aplicación de la combinación entre *Split-Screen y Scrolling* consiste en juntar una imagen compuesta por dos mitades, lo que se puede utilizar de forma extraordinaria como inicio de una demo. Para ello se controla la mitad superior, que se desplaza desde arriba hasta el centro, mediante la *Linear Starting Address*, y simplemente se desplaza mediante un scrolling vertical (en este caso hacia abajo). Para ello hay que tener en cuenta que la página de pantalla 1 o bien esté vacía o rellenada de un determinado color, ya que al principio esta visible a la mitad. La mita inferior se controla mediante una división de pantalla (splitting): reduciendo el número de línea se desplaza el comienzo de la división de pantalla hacia arriba, o sea hacia el centro. El procedimiento *Squeeze* de MODEXLIB.ASM cumple estas dos tareas:

```
squeeze proc pascal far      ;encoge pantalla
  mov si,200*80              ;valor inicial para dirección inicial
  mov di,199                 ;valor inicial para línea de Split
sqlp:                        ;bucle principal
  call waitretrace           ;esperar retrazado
  call split pascal, di      ;activar la mitad inferior con splitting
  call setstart pascal, si   ;activar la mitad superior con scrolling
  sub si,80                  ;una línea más, es decir hacia abajo
  dec di                     ;Split una línea hacia abajo
  cmp di,99d                 ;subir mitad
  jae sqlp                   ;¿listo ?
  ret
squeeze endp
```

En este procedimiento se utilizan los registros *SI y DI* para un fin un poco diferente: ya que aquí no importa la velocidad, dado que la CPU tampoco tiene demasiado trabajo y la VGA se encarga de casi todo el trabajo. Pero tampoco estaría de más utilizar las variables de registro para ahorrar memoria y tiempo de cálculo. *SI* contiene aquí el valor del registro *Linear Starting Address*, y se encarga de esta forma del movimiento de la parte superior de la pantalla. *DI* describe la línea de división, que es la que controla la parte inferior. Estos registros son cargados de entrada con sus valores de inicio, que generan una imagen que sale completamente de la pantalla.

En el bucle se coloca la situación actual mediante los correspondientes registros, siempre después de la espera obligada al retrace vertical. El movimiento se genera en las siguientes líneas, que reducen los registros en 80 (*Linear Starting Address*, 1 línea) o en 1 (*Split-Line*), consiguiendo de esta forma un scrolling hacia abajo de la parte superior y un desplazamiento de la línea de división hacia arriba, o sea que juntan la imagen. La condición de interrupción se da cuando la línea de división sobrepasa la mitad de la pantalla.

Como ejemplo sirva aquí el pequeño programa demo *Squeeze*, que simplemente carga una imagen y la junta. Mediante la espera a la tecla Return nos muestra una característica que hay que ver durante la creación de la imagen: la imagen que aparece en la pantalla en su forma original antes de que se pulse la tecla Return tiene la parte inferior y superior de la pantalla invertida, ya que debido a la línea de división se muestra en la parte inferior del monitor el principio de la memoria de vídeo, obteniendo la parte superior de la pantalla mediante el scrolling los datos de la segunda parte de la primera página de pantalla. Para mantener este formato basta girar la imagen final recortando y copiando ambas mitades. El programa se entiende por sí solo después de estas explicaciones:

```
uses Crt,ModeXLib,Gif;
Begin
  Init_ModeX;                    {activar el modo X}
  LoadGif('squeeze');            {cargar la imagen}
  p13_2_ModeX(vram_pos,rest div 4);
  ReadLn;                        {esperar un Enter}
  Squeeze;                       {«comprimir» la imagen}
  ReadLn;
End.
```

6.6 Scrolling suave en el modo texto

Un scrolling funciona en el fondo en el modo texto de la misma forma que en el modo gráfico: A través de la dirección de inicio (*Linear Starting Address)* se desplaza la parte visible de la memoria de vídeo. Pero existe un pequeño problema: este

scrolling tiene unos movimientos bastante abruptos, ya que se desplazan caracteres enteros, mientras que en el modo gráfico se desplazaban puntos individuales. Esto se debe a la estructura de la memoria de vídeo en el modo de texto: aquí ya no existen informaciones sobre los puntos individuales, de forma que la *Linear Starting Address* también hace referencia a caracteres enteros, permitiendo de esta forma solamente un scrolling quebrado.

La solución vuelve a aparecer en forma de nuevos registros de la VGA, que solamente esperan para solucionar estos problemas. Se trata en este caso de los registros de *Panning* vertical y horizontal. Panning significa el desplazamiento del contenido de la imagen en un punto, permitiendo estos dos registros que se realice esto en el modo texto.

Para conseguir ahora un scrolling suave, se desplaza simplemente el contenido de la imagen mediante un *Panning* en pasos individuales en la dirección deseada. En cuanto se ha desplazado un carácter, se restablece el valor original del registro del *Panning* y se modifica a continuación el registro *Linear Starting Address*. Esto es necesario, ya que el *Panning* solamente es posible para la altura y anchura de un carácter, y sirve por lo tanto solamente para el control detallado, mientras que el *Linear Starting Address* sigue encargándose del scrolling general.

El Panning vertical se genera mediante el registro 8 del CRTC (*Initial Row Address*, dirección inicial de fila), cuyos bits 4-0 indican con qué línea gráfica comenzará la representación de la primera línea de pantalla. Si se incrementa este valor en 1, se comenzará con la línea 1 dentro del juego de caracteres, o sea que el contenido de la pantalla se desplazará una línea hacia arriba.

Del Panning horizontal, en cambio, se encarga el controlador de atributos (*Attribute Controller*), cuyo registro *13h* (*Horizontal Pixel Panning*) se encarga de movimientos regulares en dirección X. En este registro hay que tener en cuenta una asignación de valores un poco rara: un valor 0 significa un Panning en un pixel, un 1 a 7 significa un Panning en 2 a 8 píxeles, mientras que un valor de 8 indica que no existe Panning de ningún tipo. La necesaria conversión de formato - sin lentas estructuras If - se consigue mediante un simple cálculo:

```
Uses ModeXLib,Crt;

Var x,                  {posición x en píxel}
    x_dir,              {dirección x}
    y,                  {posición y en píxel}
    y_dir:Word;         {dirección y}
```

```
Procedure Wait_In_Display;assembler;
{contrario a Wait_In_Retrace, espera construcción de pantalla}
asm
  mov dx,3dah                {Input Status 1}
@wait2:
  in al,dx
  test al,8h
  jnz @wait2                 {Display activo? -> listo}
End;
Procedure Wait_In_Retrace;assembler;
{espera retrazado, además repone el Flip-Flop de ATC a
 Input Status 1}
asm
  mov dx,3dah                {Input Status 1}.
@wait1:
  in al,dx
  test al,8h
  jz @wait1                  {Retrace activo? -> listo}
End;
Procedure FillScreen;
{llena la memoria de pantalla con la imagen en 160*50 caracteres}
var i:word;
Begin
  For i:=0 to 160*50 do Begin  {bucle de caracteres}
    If i mod 10 <> 0 Then {escribnir contador columnas?}
      mem[$b800:i shl 1]:= {no, entonces '-'}
        Ord('-') Else
      mem[$b800:i shl 1]:= {si, número col. en decenas}
      ((i mod 160) div 10) mod 10 + Ord('0');
    If i mod 160 = 0 Then {col. 0 ? -> escribir contador col.}
      mem[$b800:i shl 1]:=(i div 160) mod 10 + Ord('0');
  End;
End;
Procedure V_Pan(n:Byte);assembler;
{realiza panning vertical}
asm
  mov dx,3d4h                {CRTC Register 8 (Initial Row Adress)}
  mov al,8
  mov ah,n                   {fijar amplitud de panning}
  out dx,ax
End;
Procedure H_Pan(n:Byte);assembler;
{realiza panning vertical}
asm
  mov dx,3c0h                {ATC Index/Data Port}
  mov al,13h or 32d          {Register 13h (Horizontal Pixel Paning)}
  out dx,al                  {selec.; Bit 5(Palette RAM Address Source)}
  mov al,n                   {activar, para no apagar pantalla}
```

```
    or al,32d                   {escribir valor de panning}
    out dx,al
  End;

  Begin
    TextMode(3);                {modo BIOS 3 (80*25 cárac., Color)}
    FillScreen;                 {construit imagen de test}
    portw[$3d4]:=$5013;         {anchura pant. virtual doble (160 caract.)}
    x:=0;                       {inicializar coord. y direcc.}
    x_dir:=1;
    y:=0;
    y_dir:=1;
    Repeat
      Inc(x,x_dir);             {movimiento en direcc. x e y}
      Inc(y,y_dir);
      If (x<=0) or (x>=80*9) {invertir en los bordes}
        Then x_dir:=-x_dir;
      if (y<=0) or (y>=25*16)
        Then y_dir:=-y_dir;
      Wait_in_Display;          {esperar hasta que funcione constr. pant.}
      SetStart((y div 16 *160)    {fijar direcc. inicial (scroll basto)}
        + x div 9);
      Wait_in_Retrace;          {esperar retrazado}
      V_Pan(y mod 16);          {panning vertical (scroll fino)}
      H_Pan((x-1) mod 9);       {panning horizontal (scroll fino)}
    Until KeyPressed;           {esperar tecla}
    TextMode(3);                {modo normal de vídeo}
  End.
```

Después de establecer el modo texto y de dibujar una imagen de prueba se cambia primero nuevamente a la doble anchura virtual, estableciendo el valor del registro del offset de fila (*Row Offset) 13h* del CRTC a un valor de 320 bytes / 4 bytes = 80 (4 a causa del acceso *DoubleWord)*. El movimiento en sí -con los valores un poco diferentes- se corresponde al funcionamiento en el modo gráfico.

Una característica especial se encuentra en la división de la espera al retrace en dos partes, *Wait_in_Display y Wait_in_Retrace*. Esto se debe a que los registros utilizados aquí se conectan con una lógica de timing diferente: La *Linear Starting Address* se carga directamente por parte de la VGA después de la entrada en el retrace, de forma que en la mayoría de los casos un cambio durante el retrace aún no muestra ningún efecto en la siguiente imagen. Esto no tenía ninguna importancia hasta ahora, ya que lo único que se hacía es retardar el proceso en un retrace más. Pero ahora se añaden los registros *Panning*, cuyo efecto es inmediato cuando son modificados. La dirección de inicio se sitúa, por lo tanto, durante la construcción de la imagen, ya que en este momento tampoco juega ningún papel, de forma que en la siguiente construcción de imagen está garantizado que estará colocada de forma correcta. Pero los registros *Panning* se establecen durante el retrace, ya que su efecto es inmediato, por lo que esto debe suceder en la parte invisible de la pantalla.

El procedimiento *Wait_in_Retrace* cumple otra tarea: al ATC se accede de forma no convencional. Un Flip-Flop cambia en cada acceso de escritura a la dirección de puerto *3c0h* este puerto entre registro de datos y registro de índice. Al arrancar el programa se desconoce el estado actual del registro. Mediante un acceso de lectura al registro *Input Status 1*, como ya se realiza de por sí en el procedimiento, se coloca el puerto *3c0h* en función de índice, de forma que se consigue un estado definido.

Dado que el acceso de escritura al ATC (controlador de atributos) se efectúa inmediatamente después de la espera al Retrace, se puede partir de la base que el modo índice está activo. La única posibilidad que este puerto cambie de estado sin que se note, es que un TSR "meta mano" a través de una interrupción, algo que es poco probable y que puede ser evitado mediante un simple comando *CLI*.

Al escribir al índice del ATC es muy importante colocar siempre el bit 5 al mismo tiempo. Este bit controla el acceso a la paleta interna del ATC. En el caso de que esté borrado, la CPU obtendrá el acceso completo, y el ATC se desconectará, de forma que en el mejor de los casos (si se vuelve a activar el bit con suficiente rapidez) se produce un centelleo, aunque normalmente el ordenador se despide y ya sólo sirve un rearranque del sistema.

Lógicamente se puede volver a combinar este scrolling con una división de pantalla (split-screen), algo a lo que renunciamos aquí, ya que se opera de la misma forma que en el modo gráfico. La división de pantalla es completamente independiente del modo, ya que se programa directamente la *scan-line* física, o línea de escaneo, en la cual se dividirá la pantalla.

6.7 Limpiar el monitor de una forma diferente - una imagen que se diluye

Ya hemos nombrado varias veces el modo *double-scan* de la VGA. Esta duplicación de líneas (para representar 200 líneas gráficas en una resolución física de 400 líneas) se consigue según el BIOS que sea o bien a través del respectivo bit 7 (*DoubleScan enable)* del registro CRTC 9 (*Maximum Row Address)* o utilizando un 1 en vez de un 0 en los bits 4-0. En estos bits se encuentran en los modos de texto la cantidad de líneas de pantalla por línea de caracteres menos 1, en el modo de texto VGA por lo tanto 15, por lo cual determina el número de líneas de pantalla a las cuales se extrae la misma información de la memoria de vídeo.

A pesar de ello en el modo de texto las líneas de pantalla no son todas iguales, ya que cada vez se utiliza otra línea del juego de caracteres (fuente). Pero en el modo gráfico no existen ningún juego de caracteres, de forma que realmente se repiten cada vez los mismos datos. Un valor de 1 en estos bits se encarga de la duplicación

de líneas, de forma que en la práctica se crea una copia de cada línea. ¿Pero qué pasaría si se utilizaran valores más altos? Justo lo que nos imaginamos: se crearán más copias, los píxeles se estirarán a lo largo en dirección Y, la resolución vertical se reducirá a la mitad.

Este efecto es aprovechado por el programa *Fliess.pas*. Lo primero que hace es presentar una imagen demo, que se va estirando en la pantalla mediante un continuo incremento de los bits 4-0. Para ello no se podrán tocar los demás bits del registro, de forma que el contenido antiguo se guarda en *old9* y al escribir el registro se compara con estos valores. Todo esto se sincroniza de la forma habitual con el retrace vertical.

```
Uses Crt,Gif,ModeXLib;
Procedure Fluir;
var i,
    Old9:Byte;
Begin
  Port[$3d4]:=9;                {CRTC Register 9 (Maximum Row Adress)
selecc.}
  Old9:=Port[$3d5] and $80;     {guardar contenido antiguo}
  for i:=2 to 31 do begin       {ahorra constantes lecturas}
    WaitRetrace;                {sincronización}
    Port[$3d5]:=old9 or i;      {escribir valor}
  End;
End;

Begin
  asm mov ax,13h; int 10h End; {modo 13h (u otro modo gráfico)}
  LoadGif('beule');             {cargar imagen de fondo}
  Move(vscreen^,Ptr($a000,0)^,64000); {y a pantalla}
  ReadLn;
  Fluir;                        {provocar flujo}
  ReadLn;
  TextMode(3);                  {restaurar estado inicial de VGA}
End.
```

En principio este procedimiento también se puede aplicar al modo Texto, pero crearía imágenes poco agradables. Esto se debe a que a pesar de que la VGA duplica o triplica los contenidos de la memoria de la misma forma que en el modo gráfico, no existen informaciones sobre el juego de caracteres para estas nuevas líneas, de forma que solamente aparecen bits basura. Este efecto se limita por lo tanto al modo gráfico.

Durante la selección de una imagen apropiada habrá que tener en cuenta que en el borde superior existan por lo menos 13 líneas en negro, o que tengan otro color uniforme, ya que sino la imagen no se diluirá, sino que simplemente se estirará.

Con 400 líneas de resolución vertical y un valor máximo de 31 para el registro *Maximum Row Address* cada línea se representará 32 veces, lo cual genera 400 / 32 = 12,5 líneas visibles, que deben contener todas el mismo color, de forma que al final la imagen sea de un solo color.

6.8 Un poco más de color, por favor - barras Copper sin copiar

Los efectos especiales que hemos explicado hasta ahora se podían conseguir casi todos mediante una simple reprogramación de determinados registros de la VGA. La CPU se encargaba solamente de la ejecución, por ejemplo el constante cambio de la dirección de inicio de pantalla en un scrolling.

En este capítulo vamos a asignarle una tarea nueva a la CPU: no se va a encargar solamente de la ejecución, sino también de funciones de control. La copia sigue siendo tabú. Esto quiere decir que la CPU controlará permanentemente la VGA, y que efectuará en determinadas líneas de pantalla modificaciones en los registros de la VGA. Mediante ello se puede generar un efecto que ya era popular en la época del C-64: las barras Copper.

Se trata de unas barras horizontales en un determinado color, que se van moviendo constantemente en dirección Y hacia arriba y hacia abajo, mientras que otros efectos gráficos, como un scroll de texto, se van ejecutando al mismo tiempo o delante de ellas. De la imagen de las barras ya se puede deducir su función fundamental: la memoria de vídeo no contiene ningún tipo de datos en estos lugares determinados, estando relleno, por ejemplo, de ceros, de forma que desde aquí no pueden partir Coppers. Para cada nueva línea de pantalla se define nuevamente lo que significa el cero, de forma que una vez puede significar un rojo intenso y otras un amarillo pálido.

De esta forma se pueden representar en pantalla bastante más de 256 colores, ya que en teoría cada línea de pantalla puede contener 256 colores, que después de cada línea pueden tener otro significado. En la práctica este abanico de colores está bastante limitado, ya que dentro de un retrace no se puede utilizar una paleta entera, sino solamente una cantidad determinada, dependiendo de la velocidad del procesador. En el ejemplo de este capítulo se va cambiando solamente el color 0, de forma que ya se generan 127 colores nuevos. Sino se reprograman solamente algunos colores, esta número se puede ampliar bastante. Pero esta posibilidad ya no tiene mucho sentido ahora que existen las tarjetas Hi-Color y True-Color.

Si se va cambiando el color durante la representación de las líneas de pantalla de forma continuada, se pueden representar estructuras horizontales en forma de barras. Los demás colores a parte del 0 no se ven afectados, de forma que no existe

ningún problema para que en el primer plano vaya apareciendo un texto en movimiento, que utilice por ejemplo los colores 1-16, mientras que en los sitios donde contiene un 0 se vuelve transparente y se ven en un segundo plano las barras Copper.

De esta forma no hay que pensar en el fondo cuando se efectúe el movimiento de un texto de scroll, un fondo que se va guardando cada tanto de la forma convencional y luego se vuelve a escribir. Simplemente se copian bloques de bytes a la pantalla, que en las partes transparentes contienen un 0. La gran ventaja radica en que para la copia se puede utilizar el rápido modo de escritura 1.

Este procedimiento se puede comparar bastante bien con los *Genlock Interfaces*, los cuales muestran una imagen de televisión en determinados puntos donde la señal de vídeo contiene un determinado color (casi siempre azul). Aquí también se efectúa una mezcla a un nivel muy bajo, y no en la lenta memoria de vídeo.

¿Pero, cómo se crean las líneas de diferentes colores? Los ordenadores domésticos suelen poseer una llamada interrupción de líneas. Mediante esta se puede programar el controlador de vídeo, de forma que cuando se llegue a determinada línea de pantalla se active una interrupción, de manera que se pueda reaccionar de una manera muy rápida a este hecho y colocar un nuevo color.

Si algunas VGA aún disponen de la ventaja de una interrupciór del retrace vertical (muchas veces se activa a través de interruptores DIP), en el caso del retrace horizontal esto se vuelve bastante más complicado. A nuestro saber no existe ninguna tarjeta que soporte esta interrupción. Por lo tanto no nos queda otra solución que ir controlando constantemente el estado de la VGA e ir contando las líneas de pantalla, a fin de modificar el color en la línea deseada.

Para ello hay que crear primero un estado de salida definido, esperando a un retrace vertical. La siguiente vez que se activa *Display enable* (se puede leer en el registro *Input Status*, bit 0, *Display enable Complement)*, podremos estar seguros de que nos encontramos en la línea 0, de forma que se puede ir contando en que línea nos encontramos actualmente consultando permanentemente este bit.

Dado que la representación de una línea de pantalla dura mucho menos que la representación de toda la imagen (más o menos 30 μs, lo que da para 990 ciclos de reloj a 33 MHz de velocidad), no nos podemos permitir que se produzcan interrupciones durante la espera, ya que en muchos casos durarían demasiado.

Después de apagar la interrupción se nos plantea otro serio problema: si además se desea reproducir sonido, se producirán problemas de tiempo, y se necesitará un timing muy ajustado, a fin de evitar que la tarjeta de sonido solicite nuevos datos justo en el momento que se espera un retrace horizontal. Este problema hay

que dejarlo aparte frente al tiempo de proceso, a fin de conseguir el efecto Copper mediante una *copia* en la memoria de vídeo. Si no se ocupa toda la pantalla con líneas de retícula - Coppern -, sino por ejemplo solamente la parte superior, dispondremos en la parte inferior de la pantalla del suficiente tiempo de procesador para pasar scrolls sobre la pantalla o efectuar cálculos de sonido.

En el Timing hay que darle un valor adicional al orden de los comandos; en el corto espacio de tiempo que dura el retrace (+/- 6 s), es casi imposible efectuar el cálculo para varias barras, por lo cual se deberá efectuar antes, mientras que en el retrace en sí ya solamente se asignará el color. El siguiente programa de ejemplo *Copper*, que se compone de una parte en Pascal, *Copper.pas* y una parte en ensamblador *Copper.asm*, muestra la programación de este procedimiento:

```
Uses Crt,ModeXLib;
var y1,                    {posición y Copper 1}
    y1_dir,                {dirección y Copper 1}
    Mascara:Word;          {máscara, para superposición de coppers}

Procedure MakeCopper(y_pos1,y_pos2,overlay_maske:word);external;
{$l copper}

begin
  TextMode(3);             {funciona en TODOS los modos de vídeo}
  y1:=Port[$3da];          {conmutar ATC en modo índice}
  Port[$3c0]:=$11 or 32;   {selecc. registro 11h}
  Port[$3c0]:=255;         {color de borde 255}
  y1:=0;                   {inicio en el borde sup. de pant.}
  y1_dir:=2;               {movimiento hacia abajo}
  Mascara:=$00ff;          {primero Copper 1 (rojo) en primer plano}

  Repeat
    Inc(y1,y1_dir);        {movimiento Copper}
    If (y1<=0) or (y1>=150)     {en el borde}
      then Begin
        y1_dir:=-y1_dir;   {invertir dirección}
        Mascara:=Swap(Mascara);  {otro Copper en primer plano}
    End;
    Write('E s t e  e s  u n  t e x t o  d e  d e m o');
    MakeCopper(y1,150-y1,Mascara);   {dibujar Copper}
  Until KeyPressed;
End.
```

Primero se activa el modo de texto 3, pero se puede utilizar cualquier modo de texto o gráfico, ya que el contar las líneas de retícula es independiente del modo de la VGA. Solamente hay que tener en cuenta la diferente resolución vertical de los

modos, que se encarga de los valores de la posición Y y la dirección Y, creando barras estiradas o comprimidas. Aquí se vuelve a utilizar la resolución física, que en los modos de 200 líneas (por ejemplo, modo *13h*) se eleva a 400 líneas (*Double Scan*).

A continuación se asigna el color del marco de la pantalla a través del registro ATC *11h* a 255, ya que el Copper utilizará el color 0 para representar las barras. Si se desea ver las barras en el borde, se establecerá el color del marco igualmente a 0. Cualquier valor diferente a 0 no solamente produce la unión de imagen y Copper, sino que también alarga la duración del retrace horizontal, lo cual produce sobre todo en los ordenadores lentos una desaparición del centelleo que aparece en el borde izquierdo de la pantalla.

Dentro del bucle se mueve el Copper según el estilo ya conocido en dirección Y y se representa, ocultándose la llamada al *WaitRetrace* (espera al retrace) en el procedimiento *MakeCopper*, a fin de garantizar que el Retrace y la cuenta de líneas estén lo más cerca posible. A parte se presenta un texto como simple demostración. Este texto nos muestra lo fácil que es modificar la pantalla de forma totalmente independiente del fondo (reproducción de texto). Además se demuestra también que aún sobra suficiente tiempo de procesador para realizar otras tareas.

El procedimiento *MakeCopper* en sí se encuentra en el módulo *Copper.asm*. Al activarlo se traspasan las posiciones Y de los dos Copper (rojo y verde) y una máscara de prioridad, la cual regula la superposición de los dos Copper. El byte bajo se encarga del Copper 1 y el byte alto del Copper 2. Un valor de 0 significa que el respectivo Copper está en un segundo plano, un *0ffh* lo traslada al primer plano. Si ambos tienen el valor 0, los colores de ambos se mezclaran por superposición, mientras que una valor de la máscara de *0fffh* borra ambos Copper en las zonas donde se superponen.

En nuestro ejemplo la máscara se invierte cuando se alcanza la posición máxima, a fin de generar un movimiento circular, en el cual cuando el movimiento del Copper rojo es ascendente este pasa al segundo plano, mientras que cuando desciende pasa al primer plano. El módulo *Copper.asm* se compone básicamente del procedimiento *MakeCopper*, que seguidamente vamos a explicar:

```
extrn waitretrace:far
data segment public
  maxrow dw (?)
data ends

code segment public
public makecopper

assume cs:code,ds:data
```

```
MakeCopper proc pascal y_pos1,y_pos2,mascara_overlay:word
; dibujar dos barras Copper en posiciones y_pos1 (rojo) e y_pos2
(verde)
; mascara_overlay: 0ff00h : Copper 2 delante
;                  000ffh : Copper 1 delante
;                  00000h : fusión de ambos Copper

altura equ 88                ;altura total por Copper

  mov ax,y_pos1              ;determinar coordenada y máxima
  cmp ax,y_pos2
  ja ax_high
  mov ax,y_pos2
ax_high:
  add ax,altura              ;sumar altura
  mov maxrow,ax              ;línea máxima, a tener en cuenta

  xor cx,cx                  ;iniciar contador de líneas con 0

  call waitretrace           ;esperar retrazado para sincronizar

next_line:
  inc cx                     ;incrementar contador de líneas

  mov bx,cx                  ;calcular color 1
  sub bx,y_pos1              ;obtener posición relativa al inicio del
Copper
  cmp bx,altura/2 -1         ;2ª mitad?
  jle copper1_up
  sub bx,altura -1           ;entonces bx:=127-bx
  neg bx
copper1_up:
  or bx,bx
  jns copper1_ok             ;positivo, entonces color
  xor bl,bl
copper1_ok:
  mov ax,cx                  ;calcular color 2
  sub ax,y_pos2              ;calcular posición relativa
  cmp ax,altura/2 -1         ;2ª mitad?
  jle copper2_up
  sub ax,altura -1           ;entonces ax:=127-ax
  neg ax
copper2_up:
  or ax,ax                   ;positivo, entonces color
  jns copper2_ok
  xor al,al
copper2_ok:
  mov bh,al                  ;bl ahora tiene color Copper 1 / bh Copper 2
```

```
  mov ax,bx                 ;calcular overlay
  and ax,mascara_overlay    ;desenmascarar Copper 1 o 2
  or al,al                  ;Copper 1 preferencia
  je Copper1_detras
  xor bh,bh                 ;borrar Copper 2
copper1_detras:
  or ah,ah                  ;Copper 2 preferencia
  je Copper2_detras
  xor bl,bl                 ;borrar Copper 1
copper2_detras:

  xor al,al                 ;seleccionar color 0 en el DAC
  mov dx,3c8h
  out dx,al

  or bl,bl                  ;si Copper 1 negro -> dejar
  je bl_0
  add bl,(128-altura) / 2 ;si no aclarar para obtener brillo
bl_0:                       ;máximo
  or bh,bh                  ;lo mismo para Copper 2
  je bh_0
  add bh,(128-altura) / 2
bh_0:
;esperar retrazado horizontal y activar Copper

  cli                       ;borrar interrupciones, ya que MUY CRITICO
en tiempo
  mov dx,3dah               ;selecionar Input Status Register 1
in_retrace:
  in al,dx                  ;esperar display
  test al,1
  jne in_retrace

in_display:
  in al,dx                  ;esperar retrazado (horizontal)
  test al,1
  je in_display

  mov al,bl                 ;cargar color 1
  mov dx,3c9h               ;y fijar
  out dx,al                 ;fijar partes de rojo para Copper 1
  mov al,bh
  out dx,al                 ;fijar partes de verde para Copper 2
  xor al,al
  out dx,al

  cmp cx,maxrow             ;última línea generada?
  jne next_line
```

```
  mov dx,3dah              ;si -> terminar
wait_hret:                 ;antes de apagar, esperar retrazado,
  in al,dx                 ;sino parpadeo en la última línea
  test al,1
  je wait_hret

  xor al,al                ;seleccionar color 0 en el DAC
  mov dx,3c8h
  out dx,al
  inc dx                   ;colocar todos a 0: negro
  out dx,al
  out dx,al
  out dx,al

  sti
  ret
makecopper endp
code ends
end
```

Primero se determina la coordenada Y más grande, para establecer a partir de qué línea de retícula ya no se debe mostrar un Copper, de forma que en esa posición se puede abandonar el proceso y dedicar el tiempo de proceso a otras tareas. El contador de líneas se pone a 0 y se espera al retrace vertical, para poder comenzar también en el monitor con la línea 0.

En el bucle de líneas *next_line* se calcula en base a las posiciones Y la línea de pantalla relativa a la línea Copper superior, la cual indica el color. Si el rayo ya se encuentra en la segunda mitad, el color se tienen que volver a reducir, a fin de garantizar la simetría. Sin importar si el rayo se encuentra por encima o por de bajo de la mitad del Copper, se generará en ambos casos fuera del Copper un tono de color negativo, que en lo siguiente será capturado y reducido a 0 (negro).

Después de este cálculo *BL* contendrá el valor de color del Copper 1, y *BH* el valor del Copper 2. Estos dos valores tendrán que ser reconvertidos cuando estén superpuestos los dos Coppers, siempre en relación a la máscara *Overlay*. Esto se consigue aquí sin los lentos CMPS y en solamente dos saltos. Para ello se une el respectivo color con la correspondiente máscara, de forma que sólo importe el Copper enmascarado con *0ffh*, pasando el enmascarado con 0 automáticamente al segundo plano. Si se enmascaran ambos con 0, se mezclarán ambos. La barra enmascarada con *offh* oculta, siempre que se represente en la posición actual, o sea tenga un color diferente a 0, el otro Copper restableciendo su parte de color a 0. En este momento ya se selecciona el registro de color DAC 0, a fin de ahorrar tiempo cuando se cambie el color en sí. Tampoco es muy probable que interfiera ninguna interrupción.

Para el caso de que el Copper no ocupe su altura máxima de 128 líneas (establecida en la variable *Altura*), se suma aquí otro valor de color. Para ello se deberían recortar los componentes oscuros, y no los medios, incrementando la luminosidad del Copper a través de una adición del resto a la luminosidad máxima.

Aquí comienza la parte crítica en tiempo, algo que ya queda demostrado por el *CLI* insertado. En esto punto se espera al inicio del retrace horizontal, de la misma forma que se hace con el vertical, esperando a un *Display enable* y un posterior retrace. De esta forma se garantiza que en los ordenadores lentos, en los cuales el rayo catódico ha avanzado mientras tanto hasta el siguiente retrace, nunca se cambie durante el tiempo de representación (Display).

Ahora ya solamente hace falta colocar el *color* 0 a través del registro de valor de color del pixel (*Pixel Color Value Register)* del DAC. El componente rojo viene determinado por el *Copper 1*, el verde por el *Copper* 2, y el azul es en ambos 0. El bucle comienza de nuevo, siempre que no se haya alcanzado la última línea (determinada por la variable *maxrow*). En este caso se vuelve a esperar a un retrace y se coloca el color 0 para el resto del monitor a negro. Hay que esperar al *Retrace*, ya que si no el cambio coincidirá con la representación de la última línea, de forma que no sea representada por completo. Ahora se pueden volver a tolerar las interrupciones y finaliza el procedimiento.

6.9 Un flan en la pantalla - el Wobbler

Mediante el retrace horizontal se pueden conseguir otros efectos bastante diferentes a las barras Copper. ¿Qué pasaría por ejemplo si en vez del color se cambiara la posición horizontal de cada línea? Si se basa este efecto en una tabla senoidal, se puede cubrir toda la pantalla con un efecto ondulado, y gracias a la programación directa de los registros CRTC, esto es posible hasta en el modo texto.

La posición horizontal de una línea de retícula se controla de la mejor forma a través del registro CRTC 4. Este registro determina, tal como indica su nombre, *Horizontal-Sync-Start*, la posición en la cual comienza a actuar el retrace horizontal. Dado que el final de la sincronización se establece de forma relativa al inicio, modificando este registro lo que se consigue es desplazar la "posición" del retrace en la pantalla, pero no su longitud, algo que en todo caso debería evitarse, ya que algunos monitores son incapaces de procesar un retrace corto, representando imágenes totalmente deformadas.

La técnica del Flan se basa por lo tanto en esperar de nuevo a una determinada línea de retícula, y rellenar a continuación en cada línea el registro 4 con nuevos valores. Estos valores se extraen de una tabla senoidal, generada mediante el procedimiento *sin_gen* de la unidad *Tools* (explicada en el apartado 2.3). Lógicamente

se pueden utilizar otras funciones, pero a nuestro entender la función del seno representa bastante bien las olas del mar.

En este efecto hay que tener mucho cuidado con el timing. A diferencia de las barras Copper, la modificación de registros debería efectuarse aquí durante la representación de la imagen a través del rayo catódico. En los Copper el color no tenía ninguna importancia durante el retrace, pero sin durante el periodo de representación, por lo que aquí se cambiaba durante el retrace. Pero en el caso del Flan la posición del retrace, durante el retrace mismo o el periodo en blanco (*Blank Time*), es muy importante, mientras que el registro carece de importancia durante la representación de los datos de la imagen. El cambio deberá efectuarse por lo tanto durante el *Display enable*, invirtiendo simplemente el orden de los dos bucles de espera.

Este efecto funciona independientemente del contenido de la imagen, ya que se actúa directamente y al nivel más bajo sobre el timing de la representación de la imagen. Solamente hay que tener en cuenta los diferentes valores de registro de los diversos modos durante la creación de la tabla senoidal. En el modo de texto 3 el valor suele ser de 85, en el modo *13h* y en el modo X en cambio suele ser de 84. Esto tiene su importancia, ya que si no se tiene en cuenta, se desplazarán las zonas fijas de la pantalla hacia la derecha o hacia la izquierda, ya que vienen determinadas por el punto 0 de la tabla senoidal. Por otro lado, el valor del registro se mueve ahora dentro de ciertos valores. Si se utiliza por ejemplo un valor erróneo de 87 para el punto cero, y se le suma una amplitud de 4, el resultado quedaría fuera del rango aceptable y no produciría ningún efecto, el seno estaría aplanado en este punto, un efecto que en el ambiente HiFi se denomina *Clipping*.

Hay que decir que el registro de inicio de sincronización horizontal (*Horizontal-Sync-Start)* está protegido como todos los registros que hacen referencia al timing horizontal, por el bit 7 del registro CRTC *11h* (*Vertical Sync End*), por lo cual deberá ser borrado antes activando una desprotección del CRTC mediante un *CRTC_unprotect* (unidad *ModeXLib)*. Al final del programa se debería activar la protección del CRTC mediante *CRTC_protect*, ya que debe existir alguna razón para que este bit de protección esté colocado de forma estándar. Los dos procedimientos son muy simples y no precisan ninguna explicación adicional: *ModexLib.pas*

```
Procedure CRTC_UnProtect;
Begin
  Port[$3d4]:=$11;          {registro 11h del CRTC (Vertical Sync End)}
  Port[$3d5]:=Port[$3d5] and not $80 {borrar bit 7 (Protection Bit)}
End;
Procedure CRTC_Protect;
Begin
  Port[$3d4]:=$11;          {registro 11h del CRTC (Vertical Sync End)}
  Port[$3d5]:=Port[$3d5] or $80 {activar bit 7 (Protection Bit)}
End;
```

Como demostración sirve el programa *Wobbler*, compuesto por el programa principal *Wobbler.pas*, al cual se conecta la parte en ensamblador *Wobbler.asm* :

```
Uses Crt,Gif,ModeXLib,Tools;
const y=246;              {determinar aquí altura y posición}
      altura=90;          {también pueden ser variables}

Var Seno:Array[0..63] of Word; {tabla de senos, se rellena después}
    i:Word;               {contador temporal}

Procedure Make_Wob(wob_pos,wob_altura,wob_offset:word);external;
{$l wobbler}

begin
  TextMode(3);            {Wobbler funciona en TODOS los modos de
vídeo!}

  Draw_Ansi('db6.ans');   {cargar archivo ANSI}
  Sin_Gen(Seno,64,4,83);  {precalcular seno}
  CRTC_Unprotect;         {librar Timing horizontal}
  ReadKey;                {esperar}
  i:=0;
  Repeat
    inc(i);               {crear movimiento}
    Make_Wob(y,altura,i); {dibujar Wobble}
  Until KeyPressed;
  CRTC_Protect;           {proteger CRTC de nuevo}
End.
```

Primero se genera la imagen de prueba en el modo normal de texto 3, se crea la tabla senoidal y se desprotegen a continuación los registros CRTC 0-7 (*CRTC_unprotect*), a fin de permitir el acceso al registro 4. En el bucle, que se activa después de pulsar una tecla, la variable *y* se va incrementando, determinando esta en la siguiente llamada al procedimiento *Make_wob* el offset actual de la tabla senoidal, encargándose de esta forma del movimiento ondular. Finalmente se vuelve a colocar el bit de protección del CRTC, a fin de reactivar el mecanismo de seguridad. El procedimiento en sí *Make_Wob* se encuentra en el módulo *WOBBLER.ASM*:

```
extrn WaitRetrace:far

data segment public
  extrn seno:dataptr      ;tabla de senos
data ends


code segment public
assume cs:code,ds:data
```

```
public make_wob
make_wob proc pascal wob_pos,wob_altura,wob_offset:word
  xor cx,cx                ;contador de líneas a 0
  call waitretrace         ;sincronización con rayo catódico

next_line:
  inc cx                   ;aumentar contador líneas

  mov bx,cx                ;determinar posición en el wobbler
  sub bx,wob_pos
  mov si,bx                ;guardar para el final

  add bx,wob_offset        ;sumar offset para movimiento
  and bx,63                ;sólo valores de 0..63 (tamaño array)
  shl bx,1                 ;acceso de arrays a words
  mov bx,seno[bx]          ;obtener valor en bx

  cli                      ;borrar interrup., MUY crítico en tiempo
  mov dx,3dah              ;seleccionar Input Status Register 1

in_display:
  in al,dx                 ;esperar retrazado horizontal
  test al,1
  je in_display
in_retrace:
  in al,dx                 ;esperar display
  test al,1
  jne in_retrace

  cmp cx,wob_pos           ;¿línea deseada alcanzada?
  jb next_line             ;no -> fijar valor estándar

  mov dx,3d4h              ;CRTC-Register 4 (Horizontal Sync Start)
  mov al,4                 ;seleccionar
  mov ah,bl                ;obtener valor seno
  out dx,ax                ;y guardar

  cmp si,wob_altura        ;fin alcanzado?
  jb next_line

  mov dx,3dah
esperar:
  in al,dx                 ;esperar retrazado horizontal
  test al,1
  jne esperar
  mov dx,3d4h              ;reponer Sync Start
  mov ax,5504h
  out dx,ax
```

```
    sti                         ;permitir interrupciones
    ret
  make_wob endp

  code ends
  end
```

Al procedimiento se le traspasan los siguientes parámetros: primero la posición vertical del Wobbler y su altura en líneas de retícula, y a continuación el offset, que indica en la práctica el desplazamiento de la primera línea, que afectará a todas las demás líneas. Si se incrementa este valor en 1, la onda senoidal se desplazará una línea hacia arriba en la pantalla, de forma que con un cambio continuado se conseguirá un efecto ondulado.

En este procedimiento se espera primero, de la misma forma que en el Copper, al número de líneas traspasadas a *wob_pos*. Antes de los dos bucles de espera de la sincronización horizontal ya se calcula la posición vertical dentro del Wobbler, a fin de ahorrar tiempo más adelante. Para acceder a la tabla senoidal, primero se suma el offset de tabla recibido. El resultado debe encontrarse dentro de la tabla, por lo cual se combina con 63 (ver capítulo 2.5). A continuación se produce un acceso a la tabla compuesta por registros WORD (por ello *shl bx,1)*. El valor se guarda temporalmente en *BX*.

Ahora se espera al periodo de representación, y no a un retrace, que es lo que se desea evitar. Si la línea de inicio no se ha alcanzado aún, se sigue contando (*jb next_line)*, en caso contrario se carga el registro CRTC 4 con el nuevo valor (senoidal).

Al principio se guardó *SI*, por lo cual aún contiene la posición vertical relativa al inicio del Wobbler, pudiendo ser utilizado para una comparación, poniéndose como condición de interrupción el alcanzar la altura del Wobbler. Aquí también se deberá esperar antes de restablecer el registro a su valor original, a fin de no estropear la representación de la última línea. Se deberá tener en cuenta nuevamente que se espera a la representación de una línea.

6.10 Animaciones a tiempo real de forma fácil - efectos de paleta

La paleta ofrece posibilidades extraordinarias para modificar completamente la pantalla mediante pocas órdenes - y por lo tanto en muy poco tiempo, ya que todos los puntos de pantalla de un valor de color pueden ser convertidos a un nuevo color de un solo golpe.

Fundido a negro

El efecto más fácil que se puede conseguir de esta forma es el fundido de una imagen. De manera similar a una película se reduce en un espacio corto de tiempo la luminosidad de la imagen de su valor normal hasta cero. Esto se puede realizar de forma muy simple por medio de la paleta, reduciendo en un bucle todos los valores de color en 1, y activando después de ello la nueva paleta. Ahora se espera al siguiente retrace, y se vuelve a reducir el valor en 1, hasta que todo esté completamente negro. El procedimiento *fade_out* del módulo *ModeXLib.asm* se encarga justamente de esta tarea:

```
fade_out proc pascal far ;fundido a negro, modo pantalla no importa
local mayor:word         ;contiene el valor máximo de color
  mov mayor,63
  mov ax,ds              ;cargar segmento destino
  mov es,ax
main_loop:               ;bucle principal se recorre 1 vez por imagen
  lea si,palette         ;offset fuente y destino de la paleta
  mov di,si
  mov cx,768             ;modificar 768 bytes
lp:
  lodsb                  ;obtener valor
  dec al                 ;decrementar
  jns establecer         ;si no era negativo -> fijar
  xor al,al              ;si no 0
establecer:
  stosb                  ;escribir valor destino a paleta

  dec cx                 ;contador de bucle
  jne lp

  call waitretrace       ;sincronizar al retrazado
  call setpal            ;fijar paleta calculada
  dec mayor              ;decrementar bucle externo
  jne main_loop          ;¿aún no está listo? seguir

  ret
fade_out endp
```

Como condición de interrupción se utiliza aquí la variable *mayor*. Esta variable no es más que un contador, que cuenta hacia atrás desde 63 hasta 0. Esta función se podría imaginar como la luminosidad máxima posible de la imagen en el momento en concreto. Ya que esta es al principio de 63 (máxima de luminosidad de un componente rojo, verde o azul), la variable *mayor* se inicializa al principio con este valor. Al reducir el valor de los colores también se reduce esta luminosidad máxima, hasta que alcanza el 0, y queda garantizado que toda la imagen está en negro.

Después de cargar los punteros origen y destino dentro del bucle principal el procedimiento llega a un bucle paralelo, el cual reduce la paleta total en 1. Para ello se carga cada valor individual, se decrementa en 1 y se vuelve a escribir, no pudiéndose sobrepasar el 0.

Habiéndose preparado la paleta de esta forma, se carga mediante *Setpal*, siempre en sincronización con el retrace vertical. También es posible colocar la paleta directamente después de su cálculo, pero para ello habrá que preparar el DAC para accesos de escritura (*port [$3c8]:=0)*, y los datos tendrán que ser escritos cada vez al puerto *$3c9*. Pero esto tiene una gran desventaja: el cálculo de los nuevos valores tarda bastante -en ordenadores lentos más que el retrace-, lo que produciría centelleos en el borde superior de la pantalla. Esto se puede evitar colocando la paleta de un sola vez mediante *SetPal* (*rep outsb)*, algo que dura bastante menos tiempo. El bucle principal va siguiendo hasta que la variable *mayor* llega al valor 0, y la pantalla esta completamente en negro. Para demostrar que este efecto de paleta también funciona en el modo texto, ofrecemos el siguiente programa de demostración FADE_OUT.PAS.

```
uses crt,modexlib,Tools;
var i:word;

Begin
  For i:= 1 to 40 do
    Write('T E X T O  D E  E J E M P L O');
  GetPal;                {cargar "Palette" con la paleta DAC actual}
  ReadLn;
  Fade_out;              {apagar imagen}
  ReadLn;
  TextMode(3);           {imagen normal}
End.
```

En esta rutina hay que cargar primero la paleta desde los registros DAC a la variable correspondiente, a fin de poder trabajar con ella. En el modo gráfico, en cambio, la paleta ya suele estar en la variable *Palette*, ya que el cargador *GIF* ya la ha colocado ahí. La llamada a *Fade_out* ya se convierte en puro trámite.

En este punto se ve claramente una de las grandes ventajas de la programación directa del DAC frente a la utilización del BIOS: una acceso a la paleta en el modo texto es imposible a través del BIOS, mientras que en el modo gráfico es insoportablemente lento. Por está razón no queda más salida que la manipulación directa de los registros.

El efecto contrario - fundir desde negro

Lo opuesto de ocultar una imagen progresivamente es fundirla de la misma forma, partiendo de una imagen en negro, e incrementando la luminosidad hasta la paleta real de la imagen.

En principio esto funciona de la misma manera que la ocultación de una imagen: los valores de luminosidad de los colores se van incrementando en cada pasada - lógicamente en sincronización con el retrace vertical - en 1 o más, hasta que se alcanza el valor destino (de la paleta original de la imagen). Como condición de interrupción ya no se puede utilizar el llegar al valor 0, sino que hay que ir comparando constantemente con el valor destino.

El siguiente programa, *fade_in.pas* sirve para demostrar, tal como explicamos en el siguiente apartado, que el fundido desde negro también se puede realizar con un procedimiento más general, que puede hacer muchas más cosas.

```
uses crt,ModeXLib,Tools;
var i,j:word;
                                PalDestino:Array[0..767] of Byte;

Procedure Fade_in(ZPal:Array of Byte);
Begin
  For j:=0 to 63 do Begin       {64 pasadas, fundir completamente}
    For i:=0 to 767 do          {calcular 768 valores}
      If Palette[i] < ZPal[i]   {¿valor actual < que valor destino?}
        Then Inc(Palette[i]);   {incrementar}
    WaitRetrace;                {sincronización}
    SetPal;                     {fijar paleta calculada}
  End;
End;

begin
  ClrScr;                       {borrar pantalla}
  GetPal;                       {cargar «Palette» con paleta DAC}
  Move(Palette,PalDestino,768); {guardar paleta}
  FillChar(Palette,768,0);      {borrar paleta antigua}
  SetPal;                       {y fijar}

  Draw_Ansi('color.ans');       {cargar imagen de fondo}

  ReadLn;
  fade_in(PalDestino);          {fundir imagen a ZielPal (pal.orig.)}
  ReadLn;
  TextMode(3);                  {restaurar estado normal}
End.
```

El procedimiento *fade_in* es el núcleo de este programa. Como parámetro se le pasa la paleta destino de la imagen, que es la que deberá tener al final del fundido. Se parte de la base de que la paleta actual es negra. Dentro de este procedimiento se encuentra el bucle principal, que se pasa 64 veces (hasta que el blanco más luminoso sea visible). El bucle calcula en cada pasada la nueva paleta y la envía a la VGA. Este cálculo es muy simple: para cada valor individual de la paleta se comprueba si aún está por debajo del valor destino. En este caso se incrementa, en caso contrario se mantiene el valor existente, ya que no se puede sobrepasar.

El programa principal de *fade_in.pas* no hace nada más que generar una imagen de prueba, y luego fundirla. Antes de ello se lee la paleta actual del DAC, que más adelante será la paleta destino, y que, por lo tanto, se guarda en *Zielpal*. A continuación se puede cambiar la paleta a negro, lo cual forma el estado de salida del fundido.

El camino general - Fundido desde cualquier fuente a la paleta destino

Hasta ahora solamente hemos realizado fundidos desde una paleta hasta negro o al revés. Lo que aún falta es efectuar un fundido desde una paleta concreta a otra. Esto no parece tener mucho sentido a primera vista, ya que a parte de una deformación progresiva de los colores no se conseguirá mucho más. ¡Falso! Además de la aplicación de este procedimiento que vamos a explicar en el siguiente capítulo, se puede conseguir de esta forma un efecto muy bonito.

¿Qué le parece, por ejemplo, que, cuando una parte de la demo finaliza, la última imagen se va convirtiendo poco a poco al blanco y negro? Se podría presentar, por ejemplo, en un primer plano un texto con los créditos.

Aquí ya se ve la utilidad de este nuevo procedimiento: después de calcular la nueva paleta en blanco y negro ya solamente habrá que efectuar un fundido hacia esta paleta. El primer problema que se nos plantea es: ¿cómo se genera una paleta en blanco y negro? Por suerte el BIOS ya posee una función que se encarga de ello, aunque aquí no la vayamos a utilizar (lenta, poco flexible), pero sirve para copiar el funcionamiento.

Para convertir un color compuesto por los componentes rojo, verde y azul, al blanco y negro, habrá que sumar los tres componentes de color, y describir con este valor los tres colores en la paleta nueva. Si se mezclan los tres colores a parte iguales, siempre resultará un degradado entre blanco y negro.

Pero la pregunta es: ¿cómo se efectúa la suma? Si se muestran los tres colores con la misma proporción, el resultado no es satisfactorio, ya que el ojo humano no concibe cada color con la misma luminosidad. Un punto azul, por ejemplo, con la máxima luminosidad es bastante más oscuro que uno verde. La proporción de mezcla óptima, que garantiza una representación lo más natural posible, también se ha copiado del BIOS: se toma un 30% del componente rojo, un 59% del verde y un 11% del azul. Este simple cálculo es realizado por el procedimiento *Make_bw* de *ModeXlib.pas:*

```
Procedure Make_bw;                {reducir paleta a B/N}
Var i,sum:Word;                   {valores: 30% rojo, 59% verde, 11% azul}
Begin
  For i:=0 to 255 do Begin
    Sum:=Round(WorkPal[i*3]*0.3+WorkPal[i*3+1]*0.59+WorkPal[i*3+2]*0.11);
    FillChar(WorkPal[i*3],3,Sum); {colocar valores}
  End;
End;
```

En el bucle se calcula primero para cada color la suma y se escribe seguidamente mediante *FillChar* a los tres componentes. Al efectuar la suma se utiliza el cálculo de coma flotante de Pascal, ya que aquí lo importante no es la velocidad. El resto es un simple cálculo de porcentajes. La utilización del procedimiento de fundido se muestra en el siguiente programa de ejemplo, *fade_to.pas:*

```
uses crt,ModeXLib,Tools;
var i:word;
    origpal,
    PalDestino:Array[0..767] of Byte;

begin
  ClrScr;
  GetPal;                         {cargar «Palette» con paleta DAC}
  Move(Palette,OrigPal,768);      {guardar paleta}
  Move(Palette,Zielpal,768);      {determinar paleta destino}
  For i:=1 to 40 do begin
    TextColor(i mod 15);
    Write('T e x t o  d e  e j e m p l o  ');
  End;
  Make_bw(ZielPal);               {llevar ZielPal a B/N}
  readkey;
  fade_to(ZielPal,1);             {visualizar paleta B/N}
  ReadKey;
  fade_to(OrigPal,1);             {visualizar paleta original}

  ReadLn;
  TextMode(3);                    {restaurar estado normal}
End.
```

Aquí no se hace nada más que convertir una imagen de pantalla a color después de pulsar una tecla en una imagen en blanco y negro, y volver a darle color después de pulsar otra tecla. Para ello se carga primero la paleta actual del modo texto mediante *GetPal* en *Palette*, y se guarda esta para su posterior utilización en *OrigPal*, mientras que el cálculo al blanco y negro se guarda en *Zielpal*.

Después de haber preparado la pantalla se calcula mediante una llamada a *make_bw* la *Zielpal* hasta el blanco y negro, no cambiando nada en la pantalla, ya que solamente se ha modificado el array *Zielpal*. Después de pulsar una tecla, se efectúa la llamada a *fade_to*. A este procedimiento se le traspasa simplemente la paleta destino que debe dar el resultado, siendo necesario que la paleta mostrada actualmente por la VGA se corresponda con la variable *Palette*.

Como parámetro adicional, *fade_pal* aún necesita la distancia de paso para el fundido. Esta corresponde a la velocidad con la que se realizará todo el proceso: en cada generación de la imagen se incrementa reduce cada valor de color con el valor traspasado; o sea, que cuanto mayor sea este valor, más rápido se realizará el fundido. El procedimiento en sí *fade_to* se encuentra en el módulo *ModeXlib.asm* de la unidad *ModeXlib*:

```
fade_to proc pascal far destinopal:dword, laenge:word, paso:byte
;funde «Palette» a «Pal_dest», paso de Pascal como array of Byte !
local mayor:word
  mov ax,63              ;calcular número de iteraciones para
  div paso               ;obtener 63
  xor ah,ah
  mov mayor,ax           ;fijar número de iteraciones
next_frame:
  les di,destinopal      ;obtener Offset, Pascal pasa los Arrays far
  lea si,palette         ;obtener offset de «Palette»
  mov cx,768             ;procesar 768 bytes

weiter:
  mov al,[si]            ;obtener valor de la paleta actual
  mov ah,[di]            ;obtener valor de ZielPal

  mov bl,ah
  sub bl,al              ;diferencia con el valor destino
  cmp bl,paso            ;más de un paso por encima?
  jg arriba              ;-> decrementar
  neg bl                 ;diferencia
  cmp bl,paso            ;mayor que paso negativo
  jg runter

  mov al,ah              ;destino alcanzado, fijar definitivamente
```

```
escribir:
    dec cx              ;dec. bucle de color
    je listo            ;0 ? -> listo
    mov [si],al         ;escribir valor a paleta
    inc si              ;seleccionar siguiente valor
    inc di
    jmp weiter          ;y seguir
runter:
    sub al,paso         ;decrementar
    jmp escribir
arriba:
    add al,paso         ;incrementar
    jmp escribir
listo:                  ;paleta calculada
    call waitretrace    ;sincronización
    call setpal         ;fijar paleta
    dec mayor           ;las 63 pasadas listas?
    jne next_frame      ;no -> seguir

    ret
fade_to endp
```

Primero se debe determinar el número de pasos por el bucle que hacen falta para garantizar que cada color llega a su valor destino. Como caso extremo nos deberemos imaginar que un color debe fundirse desde 0 (componente de un color no existente) hasta 63 (máxima luminosidad de un componente de color). En un unidad de paso de 1 harán falta 63 pasos, por el contrario, con una unidad de paso de 2 solamente la mitad. Por esta razón, el contador de bucle *mayor* se inicializa con 63/*paso*.

A continuación se carga *ES:DI* con el puntero a la paleta recibido de Pascal. Aquí solamente interesa el offset, ya que la paleta está situada de por sí en el segmento de datos, pero no es demás cargar también el registro *ES* (si no opina que *mov di, destinopal* +2 es más elegante).

El bucle *weiter*, que es ejecutado 768 veces (256 colores * 3 componentes), carga primero *AL* con el valor actual del color que se está evaluando actualmente, y *AH* con el valor destino que se desea alcanzar. Ahora solamente falta determinar la dirección en la que se desea efectuar el fundido: hacia arriba, o sea incrementando la luminosidad, o hacia abajo, reduciéndola. Pero una simple comparación no sirve para nada, como demuestra el siguiente caso: imagínese que el valor inicial es 0, mientras que el valor destino es 15, y se utiliza una unidad de paso de 2. Después de 7 pasos por el bucle el valor actual será de 14, o sea demasiado bajo, por lo que deberá ser incrementado. Esto daría un resultado de 16, que vuelve a ser demasia-

do alto, de forma que se vuelve a reducir a 14. El registro de la paleta va cambiando de un valor a otro sin llegar nunca al valor deseado. Esto produce efectos poco estéticos, sobre todo en fundidos rápidos donde la anchura de paso es muy alta, ya que los saltos de esta fluctuación también serán muy altos.

La solución consiste en no efectuar la comprobación en base a una igualdad, ni a un mayor que o menor que, ya que se funde en ambas direcciones, sino en comprobar si el valor actual ya se ha acercado en menos de un paso al valor destino. Para ello se crea simplemente la diferencia (en el registro *BL)*. Si ésta es positivamente mayor (*JG, Jump if greater*, saltar si es mayor) que la anchura de paso, el valor aún estará lejos del valor destino, por lo cual deberá ser reducida. Si la diferencia en cambio es menor (superior en valor) que la anchura de paso negativa, deberá incrementarse. Si no se cumple ninguna condición, se habrá alcanzado el valor destino, y ambos valores se podrán igualar, a fin de corregir la última y ligera diferencia.

No tiene ninguna importancia si en la etiqueta se resta la anchura de paso, si se suma al valor actual o si se igualan ambos valores, todos estos caminos llevan a la etiqueta *escribir*, la cual escribe el nuevo valor en la paleta y desplaza el puntero. En cuanto se hayan procesado los 768 valores, se efectúa un salto desde el bucle interior a la etiqueta *listo*. Aquí se coloca, después de una sincronización, la paleta nueva, y siempre y cuando *mayor* aún no sea 0, se calcula la siguiente paleta (*next_frame* en relación a la siguiente generación de imagen).

Como en las películas - Fundido de una imagen a otra

Ya es bastante más profesional el cambiar de una imagen a otra haciendo desaparecer la primera imagen y haciendo aparecer la segunda, en vez de efectuar simplemente un cambio brusco entre ambas; esto ya lo puede realizar con los procedimientos explicados anteriormente. Pero para conseguir un cambio de imagen realmente profesional es inevitable conseguir un cambio enlazado de una imagen a otra, o sea que se deben mezclar ambas imágenes.

Para conseguirlo existen hoy en día una gran cantidad de programas llamados de *Morphing*, pero todos ellos tienen la desventaja de trabajar con imágenes ya preparadas. Estas tienen que ser calculadas con mucho esfuerzo, ocupan mucho espacio en el disco duro y en la memoria y tienen que ser copiadas de forma complicada (y lenta) a la memoria de vídeo. Se trata por lo tanto de una solución para principiantes, mientras que los profesionales calculan sus fundidos enlazados a tiempo real.

En el caso de imágenes de paleta -todos los modos gráficos tratados hasta ahora se basan en paletas- el fundido se convierte en el fondo en un simple fundido de paletas, tal como mostraremos a continuación (mediante el procedimiento *fade_to*

del apartado anterior). Solamente en el caso de complicadas metamorfosis, las cuales mezclan al mismo tiempo partes de imágenes mediante movimientos, habrá que utilizar el software de *morphing* existente.

Lo que se realiza, por lo tanto, es fundir una paleta hacia otra. Pero dado que ambas imágenes están colocadas una encima de otra, y cualquier cambio de los datos de la imagen (que es lo que hace el morphing) es imposible a causa de su poca velocidad, los mismos datos de la imagen deberán representar diferentes imágenes según la paleta que se utilice. Primero está activa la paleta origen, representando los datos la imagen original. Al final está activa la paleta destino, donde los mismos datos de imagen que antes representaban la imagen original deberán crear ahora la imagen destino. De los pasos intermedios se encarga el procedimiento *fade_to*. Durante un proceso de este tipo se nos plantean dos preguntas:

1. ¿Cómo se deberán manipular las imágenes para que -según la paleta- representen tanto la imagen original y la final?
2. ¿Cómo deben ser las paletas para poder generar diferentes imágenes a partir de los mismos datos?

En cuanto a la pregunta 1: En un fundido de este tipo nos encontramos ante un problema completamente nuevo. En los fundidos explicados hasta ahora o bien el color destino era el mismo para todos los puntos, o sea negro, o el color origen. En cambio ahora, habrá que convertir cualquier color en cualquier otro color. Existen puntos rojos que en la imagen destino deberán ser verdes, pero también rojos que al final deberán ser azules.

Aquellos de los lectores que eran buenos en matemáticas (estocástica, combinaciones) sabrán lo que significa esto para un abanico de colores: al combinar un número determinado de colores con el mismo número de colores diferentes existen tantas combinaciones posibles como el número de colores elevado al cuadrado. Durante el fundido hay que tener en cuenta todas estas combinaciones, por lo que deberán tener un registro en la paleta utilizada durante el fundido. Para fundir una imagen de dos colores en otra se utilizarán por lo tanto *cuatro* registros de paleta (color 0 a color 0, o sea igual, color 0 a color 1, 1 a 0 y 1 a 1). Si se utilizan dos imágenes de cuatro colores ya se necesitarán 16 registros.

Pero dado que la VGA solamente dispone de una paleta con 256 colores, el número máximo de colores que se puede fundir sin modificar los datos de la imagen es de 16. En los fundidos perfectos sin centelleos este número aún se reduce a 15, tal como explicaremos más adelante.

Basándonos en estas combinaciones ya podemos deducir el procedimiento que se aplicará para la mezcla de las imágenes: cada combinación deberá estar represen-

tada, utilizándose para ello *N* bloques con *N* registros, donde *N* representa el número de colores por imagen. El número de bloque corresponde al color destino, mientras que el índice dentro del bloque corresponde al color origen. También sería posible al revés, pero complicaría sobretodo el reset que explicaremos luego. Para determinar el número de color, se utiliza simplemente la siguiente fórmula:

```
Número de color := Color destino * cantidad de colores + color origen
```

Este enunciado ya es famliar: de la misma forma se calcula un byte hexadecimal para convertirlo en un número decimal: *Nibble superior * 16 + Nibble inferior*. Esto aceleraría bastante el proceso cuando se utilicen 16 colores, ya que simplemente se debería cargar el color destino en el nibble superior y el color origen en el inferior. Pero aquí se plantea el problema de que no se pueden utilizar 16 colores, sino como máximo 15, por lo que deberemos seguir con las lentas multiplicaciones. Esto tampoco tiene mucha importancia, ya que la parte del programa que contiene las multiplicaciones no es crítica en cuanto a tiempo, y como máximo producirá un ligero retardo del fundido en sí. El siguiente diagrama muestra la formación de los bloques cuando se desean fundir dos imágenes a cuatro colores:

Color 0	Color 1	Color 2	Color 3	Color 4	Color 5	Color 6	Color 7	Color 8	Color 9	Color 10	Color 11	Color 12	Color 13	Color 14	Color 15
Color origen 0	Color origen 1	Color origen 2	Color origen 3	Color origen 0	Color origen 1	Color origen 2	Color origen 3	Color origen 0	Color origen 1	Color origen 2	Color origen 3	Color origen 0	Color origen 1	Color origen 2	Color origen 3
Color destino 0				Color destino 1				Color destino 2				Color destino 3			
Bloque 0				Bloque 1				Bloque 2				Bloque 3			

Figura 7: Paleta de colores para fundidos enlazados

Este esquema responde de paso a nuestra segunda pregunta sobre la organización de las paletas: la generación de los bloques se corresponde exactamente con la estructura de la paleta: La paleta origen (de tamaño 1 bloque) se copia tantas veces como corresponde al número de colores -de esta forma se crean exactamente el mismo número de bloques de origen-, mientras que la paleta destino está ampliada a su tamaño completo. Para ello se multiplica cada color de forma individual, de forma que los bloques de esta paleta sean homogéneos.

En este efecto no es necesario utilizar varias páginas de pantalla, o crear páginas precalculadas y copiarlas a la pantalla, ya que la manipulación se puede efectuar directamente en la pantalla. Habrá que tener en cuenta que la imagen que realmente está visible no cambie durante las modificaciones (o sea durante la generación de la paleta y la mezcla), ya que en caso contrario se producirían centelleos en la pantalla. Si la imagen no se puede modificar, habrá que tener en cuenta dos puntos: la paleta que está activa en este momento solamente podrá ser modificada

en aquellos colores que actualmente no estén en pantalla, y por otro lado cuando se modifiquen los datos de la imagen la paleta ya deberá estar preparada, para que los puntos modificados vuelvan a coincidir con su color original. Si se va a situar por ejemplo un punto rojo, que tiene el color 3, al color 23, habrá que asegurarse que el color 0 ya contiene el rojo, a fin de que en la pantalla no cambie nada. De todo ello se deduce el orden de las llamadas a los procedimientos que se debe cumplir durante un fundido encadenado:

1. Preparar las paletas, modificando solamente los colores inactivos.
2. Mezclar los datos de imagen, no siendo visible ninguna modificación.
3. Fundir las dos paletas.

La exigencia de modificar solamente las zonas inactivas de la paleta obliga a la introducción de un bloque de colores adicional. Este bloque contiene primero los colores origen que son utilizados por la imagen origen. Dado que una imagen de cuatro colores suele ocupar los colores 0 a 3, este nuevo bloque se encontrará en esta posición. Para generar la paleta origen (en este caso copiar el bloque cuatro veces) este bloque se utiliza como paleta origen y se multiplica, sin modificarse para ello el bloque en sí, de forma que en la pantalla no pasa nada.

Este bloque, que por cierto se llama *Reset-Block*, tiene que cumplir otra tarea importante: después de haber efectuado un fundido la imagen visible tendrá una paleta mucho más amplia que las imágenes originales, justo el número de colores de éstas elevado al cuadrado. Si ahora se desea fundir otra imagen, se planteará el problema de que se utilizan demasiados colores, ya que como máximo deben ser 15. Aquí se aprovecha que los bloques destino son homogéneos, lo que quiere decir que todos los colores de un bloque tienen el mismo contenido, por lo cual pueden ser sustituidos sin ningún tipo de problema. En el fondo (en el ejemplo explicado) solamente existen cuatro (!) colores diferentes en la pantalla.

Para reducir todo este número de colores a una paleta que permita un fundido se vuelve a utilizar un *Reset-Block*, que de por sí ya contendrá la paleta original. De ahí el nombre de *Reset-Block*: los datos de imagen se reducen a este bloque, o sea que se efectúa un reset. En este bloque se encontrará después de efectuar el fundido la paleta destino, de forma que ya solamente se deberán calcular los datos de la imagen.

Este cálculo es muy simple, ya que solamente hay que invertir del proceso para conseguir el color destino: una división entre el número de colores basta para obtener el número de bloque y por lo tanto el color destino, debiéndose restar, como en todas las operaciones de este tipo, el *Reset-Block*.

La existencia de este bloque es la causa de que solamente puedan utilizarse 15 colores: a parte de los registros de paleta necesarios para el fundido (n^2, n= cantidad de colores utilizados) también se necesita un *Reset-Block*, que también deberá tener el tamaño de n colores, de forma que para fundir dos imágenes se precisan $n*n+n$ o $(n+1)*n$ registros de paleta. En el caso de 16 colores resultarían $16*17 = 272$ valores de paleta, que, por desgracia, la VGA no posee. Por esta razón solamente se pueden fundir imágenes que tengan un máximo de 15 colores, precisándose $15*16 = 240$ *colores*.

Muchas veces tiene sentido limitar aún más el número de colores, a fin de liberar en el borde superior de la paleta los colores que no se utilizan, que pueden ser utilizados para partes estáticas de la imagen, por ejemplo para un logotipo fijo debajo del cual se funden textos. En este caso habrá que utilizar para la parte estática de la imagen valores de color del extremo superior de la paleta, ya que para el fundido se utilizan los inferiores.

Al mezclar ambas imágenes se calcula, según la fórmula explicada antes, el nuevo registro de la paleta y se vuelve a escribir a la RAM de la VGA. Aquí deberemos recordar lo siguiente: prácticamente sólo se escribe en otro número de bloque, sin que cambie el índice dentro del bloque (correspondiente al color origen). Dado que todos los bloques tienen en este momento el mismo contenido, en la pantalla aún no cambia nada.

Finalmente solo faltará fundir la paleta actual, conseguida a través de la multiplicación de la paleta original, con la paleta destino, de forma que después los bloques sean homogéneos y contengan la paleta destino, de forma que sea visible la imagen destino. Las modificaciones efectuadas no serán visibles hasta que se efectúe el fundido en sí, de forma que este punto es el único en el que habrá que vigilar la velocidad, utilizándose aquí el procedimiento *fade_to*, el cual como sabemos no tiene ningún problema de velocidad. Para cada generación de imagen habrá que volver a cargar los registros de la paleta, lo cual es mucho más rápido que si se modificarán los datos de la imagen en sí. Para adquirir un poco de práctica, sigue a continuación una aplicación de fundido enlazado. Este programa, *fade_ove.pas*, utiliza la unidad *Fade.pas*, que describimos a continuación:

```
uses Crt,ModeXLib,gif,fade;
Var pic1_pal,                    {paletas de ambas imágenes}
    pic2_pal:Array[0..767] of Byte;
    pic1,                        {contiene 1ª imagen}
    pic2:Pointer;                {2. imagen, igual a vscreen}

Begin
  Init_Mode13;                   {activar modo 13h}
  Screen_off;                    {apagar pantalla durante carga}
  LoadGif('schach');             {cargar primera imagen}
  GetMem(pic1,64000);            {obtener memoria para 1ª imagen}
```

```
  Move(vscreen^,pic1^,64000);   {guardar en pic1}
  Move(Palette,pic1_pal,768);   {y guardar la paleta}
  Show_Pic13;                   {esta imagen en pantalla}

  LoadGif('kiste');             {cargar la siguiente en vscreen^}
  pic2:=vscreen;                {pic2 como puntero a ella}
  Move(Palette,pic2_pal,768);   {guardar su paleta}

  Move(pic1_pal,Palette,768);   {activar paleta de imagen 1}
  SetPal;                       {y fijar}
  Screen_on;                    {activar pantalla de nuevo}

  ReadLn;                       {esperar}
  Fundir(pic2,pic2_pal,0,0,200);
                                {y mostrar imagen 2)}

  fade_ResetPic(0,200);         {preparar nuevo fundido}
  ReadLn;
  Fundir(pic1,pic1_pal,0,0,200);
                                {y fundir imagen 1}

  ReadLn;
  TextMode(3);
End.
```

En este programa se funden dos imágenes GIF completamente diferentes. Para ello se precisan las siguientes variables: *pic1* contiene un puntero a los datos de la primera imagen en la memoria principal (primero hay que alocar este puntero), *pic2* señala de forma correspondiente a la segunda imagen, aprovechándose aquí que ya existe un bloque del tamaño correspondiente en la memoria principal (*vscreen*). A fin de mantener un cierto orden se sigue utilizando el puntero *pic2* como sinónimo. Las dos paletas individuales se guardan en las variables *Pic1_Pal* y *Pic2_Pal*.

Después de activar el modo *13h* primero se oscurece la pantalla, ya que la paleta se cambia durante la carga y no es aconsejable que el usuario vea estos procesos. Se carga la primera imagen y se desplaza a la dirección *Pic1* (después de direccionarla); su paleta se guarda en *Pic1_pal*.

Después de copiar esta primera imagen a la RAM de la VGA, se carga la segunda. Esta imagen se encontrará durante el resto del programa en *Vscreen*, utilizándose como sinónimo el puntero *Pic2*, que debe asignarse aquí. Después de guardar la segunda paleta en la correspondiente variable, se activa la primera paleta (copiada a *Palette*), y se mueve a la VGA (*setpal*). Ahora se puede volver a encender la pantalla.

El fundido en sí es realizado por el procedimiento *Fundir*. A este procedimiento hay que pasarle como parámetros: la dirección de la imagen destino, su paleta, la coordenada Y dentro de la imagen destino, la coordenada Y a la cual se copiará la imagen destino y la altura de la imagen destino. Estos últimos tres valores no tienen ninguna importancia en este programa, lo único que hacen es describir una copia de la imagen completa. Los primeros dos parámetros se nutren con los valores de la segunda imagen, de forma que en el transcurso del procedimiento se funde hacia esta imagen.

Después de efectuar el fundido se restablece la imagen, a fin de posibilitar un nuevo fundido. Aquí también se traspasan las coordenadas Y dentro dela RAM de la VGA, así como la altura, aunque aquí carezcan de importancia. Finalmente se vuelve a fundir hacia la primera imagen y el programa finaliza. La orden central de este programa es la llamada a *Fundir*. Este procedimiento unifica todas las tareas necesarias para el fundido y se encuentra en la unidad *Fade.pas*:

```
Unit fade;
{emplea una imagen nueva para fundir con la que se muestra
 actualmente}

Interface
Uses ModeXLib;

Var Colors:Word;                {nº colores por imagen}

Procedure fade_ResetPic(y,Altura:Word);
Procedure Fundir(Pic:Pointer;Pal:Array of Byte; Start,y,Altura:Word);


Implementation
Var i,j:Word;                   {contador temporal}
                                Pal_Destino:Array[0..768] of Byte;
{paleta destino temporal}

Procedure fade_set(Fuente:Pointer;Start,y,Altura:Word);external;
{«mezcla» fuente con VGA-Ram}
{donde fuente emplean Altura en la altura para línea Start y VGA-Ram
desde y}

Procedure fade_ResetPic(y,Altura:Word);external;
{prepara imagen fundida para un fading nuevo}
{reducción de "Colors^2" a "Colors" colores}
{también aquí y=línea en VGA-Ram, Altura=altura de la zona a proce-
sar}

{$l fade}
```

```
Procedure fade_CopyPal;
{multiplicar paleta a Colors^2 (multiplicar bloque 0 no homogéneo)}
Begin
  For i:=1 to Colors do
    Move(Palette[0],Palette[i*3*Colors],Colors*3);
End;

Procedure fade_expandir(Var Pal:Array of Byte);
{abrir paleta a Colors^2 (multiplicar cada color individual)}
{aquí se forman los bloques homegéneos de 0..Colors-1}
Begin
  For i:= 0 to Colors-1 do        {procesar cada color indiv.}
    For j:=0 to Colors -1 do      {escribir Colors veces}
      Move(Pal[i*3],Pal[(i+1)*3*Colors+j*3],3);
End;

Procedure Fundir(Pic:Pointer;Pal:Array of Byte; Start,y,Altura:Word);
{funde de la imagen visible actualmente a pic (con paleta Pal)
 se comienza en línea «Start» de Pic, copiando «Altura» líneas a la
 coordenada y de la imagen actual}
Begin
  WaitRetrace;                    {sincronización}
  fade_CopyPal;                   {multiplicar bloque en paleta actual}
  SetPal;                         {fijar de nuevo la paleta}
  Move(Palette,Pal_Destino,768);        {mantener partes orig de paleta}
  Move(pal,Pal_Destino,Colors*3);       {cargar paleta destino}
  fade_expandir(Pal_Destino);     {extender bloques de paleta destino}
  fade_set(pic,start,y,altura);   {mezclar nueva imagen}
  fade_to(Pal_Destino,1);         {y fundir}
End;

Begin
  Colors:=15;                     {¡sólo valor por defecto!}
End.
```

Como variable global esta unidad utiliza *Colors*. En ella se indica cuántos colores utiliza cada una de las imágenes. Si, por ejemplo, se funden dos imágenes de 15 colores, esta variable deberá contener el valor 15, el cual es al mismo tiempo el valor por defecto.

Todo el proceso de fundido es controlado por el procedimiento ya nombrado *Fundir*. A fin de evitar con toda seguridad centelleos en la pantalla, de entrada se espera al Retrace. A continuación se prepara la paleta original, que es la paleta actual (en la variable *Palette*), para el fundido mediante una multiplicación y finalmente se coloca (no se produce ninguna modificación en la pantalla, ya que solamente se modifican registros de paleta inactivos).

Mediante las posteriores ordenes *Move* se genera la paleta destino. Se toma la paleta actual como paleta de salida, para que las partes estáticas de la imagen, que utilizan los registros de paleta situados al final de la misma, también estén presentes en la paleta destino. Además se añade a la parte inferior la paleta recibida en *pal* (registros *Colors* a 3 bytes). Esta zona inferior se estira mediante *fade_expandir* y se prepara de esta forma para el fundido.

Mediante *fade_set* se mezclan finalmente los datos de las imágenes en sí, de forma que, según la paleta, representen o bien la imagen origen o la destino. Al final se arranca el proceso de fundido, fundiéndose mediante el procedimiento ya conocido *fade_to* hacia la paleta destino, apareciendo lentamente la imagen destino en la pantalla. La unidad posee dos procedimientos internos en Pascal *fade_copyPal y fade_expandir*. En este caso no importa la velocidad, por lo que estos procedimientos se mantienen en Pascal.

Fade_copyPal prepara la paleta origen (contenida en *Palette)* para el fundido, multiplicando el bloque reset (colores 0 a *Colors -1)* con *Colors*. El bucle no hace otra cosa que copiar los *Colors* por *Colors * 3 bytes* a la posición del bloque.

El procedimiento *Fade_expandir* es un poco más complicado: aquí se debe ampliar cada color a un bloque. Para ello se precisan dos bucles anidados. Un bucle exterior (*Contador i*) se encarga que cada color (desde 0 hasta *Colors -1)* sea ampliado a la anchura necesaria. Dentro de este bucle se encuentra el bucle *j*, que genera copias de cada color *Colors*, que se sitúan una al lado de otra dentro de un bloque. De esta manera se rellena el bloque que se corresponde con el color tratado con ese color. La orden *Move* se encarga de este relleno, copiando el color activo (en la posición de paleta *i*3)* a cada posición *(j*3)* dentro del bloque actual *((i+1)*3*Colors)*. Aparte de utilizar los procedimientos Pascal la unidad también utiliza dos procedimientos en ensamblador del módulo *fade.asm: fade_set y fade_ResetPic*.

```
data segment public
  extrn colors:word
data ends

code segment public
assume cs:code,ds:data
public fade_set,fade_ResetPic


col db 0                    ;equivalente de segmento de código a Colors


fade_set proc pascal near fuente:dword,start:word,y:word,altura:word
```

```
    mov ax,colors              ;guardar colores en var. de segm. código col
    mov col,al
                               push ds
    mov ax,word ptr Fuente + 2  ;puntero fuente a ds:si
    mov ds,ax
    mov si,word ptr Fuente

    mov ax,320                 ;añadir direcc. inicial de la imagen fuente
    mul start
    add si,ax

    mov ax,0a000h              ;puntero 0a000:0 a es:di
    mov es,ax
    mov ax,320                 ;añadir direcc. inicial de la imagen destino
    mul y
    mov di,ax

    mov ax,320                 ;convertir altura en nº de bytes
    imul altura
    mov cx,ax

  lp:                          ;bucle principal
    lodsb                      ;valor destino en al
    mul col                    ;calcular nuevo valor de color
    add al,es:[di]             ;añadir valor actual
    add al,col
    stosb                      ;y escribir

    dec cx                     ;¿todos los puntos copiados?
    jne lp

    pop ds
    ret
  fade_set endp

  fade_ResetPic proc pascal far y:word, altura:word
    mov ax,0a000h              ;dirección VGA 0a000:0 a es:di
    mov es,ax

    mov ax,320                 ;respetar línea y
    mul y
    mov di,ax

    mov ax,320                 ;calcular nº de bytes a procesar
    mul altura
    mov cx,ax
  res_lp:
    mov al,es:[di]             ;obtener valor
    xor ah,ah                  ;¡borrar ah en división!
```

```
    div byte ptr colors     ;calcular nº de bloque
    dec al                  ;quitar Reset-Block
    stosb                   ;escribir

    dec cx                  ;¿todos los puntos acabados?
    jne res_lp              ;no, seguir

    ret
fade_ResetPic endp

code ends
end
```

Estos procedimientos también acceden a la variable global *Colors*, que viene indicada aquí en el segmento *data*. Pero dado que temporalmente el procedimiento *fade_set* no puede acceder a esta variable, a causa de la modificación de *DS*, se utiliza para ello la variable de segmento de código *col*.

El procedimiento *fade_set* copia primero el contenido de *Colors* a *Col*, liberando de esta forma *DS* a fin de poder recibir el puntero destino (parte del segmento). A continuación se carga *SI* con el offset de la imagen destino (datos de imagen). Aquí se debe tener en cuenta la coordenada de inicio recibida, la cual indica la línea a partir de la cual se leerá la imagen destino. El offset de esta línea se calcula mediante una multiplicación por 320 y se suma a *SI*.

Ahora se carga el puntero *ES:DI* con la dirección de la VGA. El segmento es simplemente el segmento de inicio de la RAM de la VGA, y el offset se calcula de la misma forma que en la imagen destino mediante una multiplicación de la coordenada Y por 320.

A *CX* también se suministra un valor apropiado, multiplicando la altura por 320. A continuación el procedimiento entra en el bucle principal *LP*.

Aquí se lee primero el valor destino de la imagen destino (*DS:SI*). Este se corresponde con el número de bloque que se deberá utilizar, por lo que deberá ser multiplicado por el número de colores. Como Offset dentro del bloque se lee además el color de la imagen origen desde la memoria de vídeo y se suma. Este valor se vuelve a escribir y se finaliza el bucle decrementando *CX* y un salto condicional.

El segundo procedimiento de la parte en ensamblador *fade_ResetPic* sirve para reducir después del fundido la imagen visible a los colores *Colors*. Los colores deberán ser reducidos mediante el número de bloque, que se corresponde con el color destino, hasta el bloque Reset con los números de color 0 hasta *Colors -1*.

Se vuelve a situar un puntero en la RAM de la VGA en la correspondiente coordenada *ES:DI*, y se suministra a *CX* un valor basado en la altura para determinar la cantidad de pasos por el bucle.

El bucle principal de este procedimiento *res_lp* lee cada vez un valor de la memoria de vídeo, calcula su número de bloque mediante una división entre el número de colores y vuelve a escribir el valor. Aquí hay que tener en cuenta nuevamente el bloque reset: este se encarga de que los números de bloque se muevan entre 1 y *Colors*, pero tienen que ser reducidos a sus valores de color reales entre 0 y *Colors -1*. Finalmente finaliza el bucle y el procedimiento.

En el ejemplo utilizado se ha fundido una imagen completa de 15 colores en otra, utilizándose (casi) toda la paleta de colores. Si se reducen los colores a fundir a, por ejemplo, 14, se obtendrán en la parte superior de la paleta colores que no se ven afectados por el fundido. Estos colores pueden ser utilizados para mostrar partes estáticas de la imagen que no cambian durante el proceso de fundido.

De esta forma se pueden mostrar, por ejemplo, los créditos de una demo: como imagen estática se muestra para cada parte de la demo una pequeña imagen (como una pantalla capturada), mientras que en la parte inferior de la pantalla se van fundiendo textos con los créditos. El funcionamiento se muestra en el programa *fade_txt.pas:*

```
Uses Crt,Gif,ModeXLib,Fade;
Var
  Text_Pal:Array[0..767] of Byte;
  i:word;

Begin
  Init_Mode13;                    {emplear modo 13h}
  Screen_Off;                     {apagar pantalla durante la carga}
  LoadGif('vflog210');            {cargar parte estática}
  Move(Palette[210*3],            {guardar parte de paleta}}
    Text_Pal[210*3],46*3);        {(colores 210..255)}
  Show_Pic13;                     {copiar imagen estática a VGA}

  LoadGif('textos');              {cargar imagen con textos}
  Move(Palette,Text_Pal,14*3);    {guardar su parte de paleta}
                                  {colores (0...13)}
  Move(Text_Pal,Palette,768);     {fijar paleta acabada}
  SetPal;

  Move(vscreen^,                  {1er texto se copia directamente}
    Ptr($a000,160*320)^,19*320);{en pantalla}
  Screen_On;                      {ahora imagen lista-> activar}
  Colors:=14;                     {¡imágenes con 14 colores!}
```

```
  For i:=1 to 6 do Begin          {ver por orden los otros 6 textos}
    Delay(500);                   {tiempo para leer}
    Fundir(vscreen,               {fundir siguiente imagen en posición}
      text_pal,i*20,160,19);      {anterior (y=160)}
    Fade_ResetPic(160,19);        {y «resetear»}
    If KeyPressed Then Exit;      {interrúmpase en cualquier momento}
  End;

  Readln;
  TextMode(3);
End.
```

Este programa carga primero la parte estática en la pantalla y escribe su parte de la paleta total a la variable *Text_pal*. A continuación se carga la imagen con los textos en *vscreen* y se copian sus partes de paleta a *Text_Pal*. Finalmente se carga y se coloca esta paleta.

Después se copia directamente la primera parte del texto a la memoria de vídeo, ya que aquí aún no hace falta un fundido. Después de volver a conectar la imagen, comienza el bucle principal. Para cada uno de los siguientes seis textos se efectúan estos pasos: primero se espera medio segundo, a fin de que el espectador pueda leer. A continuación se funde el texto, utilizándose las coordenadas en la llamada a *Fundir*: de la imagen destino se copia a partir de la línea *j*20*, dado que los textos comienzan en este ejemplo en las coordenadas Y 0,20,40 etc. Todos tienen una altura de 19 puntos y se funden en la línea de pantalla 160.

En la posterior llamada a *Fade_resetPic*, se vuelven a utilizar de nuevo estas coordenadas. En último lugar, se realiza una llamada al teclado, permitiendo, de esta manera, una interrupción del proceso.

Durante el desarrollo de imágenes propias para este efecto hay que tener en cuenta la correcta distribución de la paleta. En las imágenes que se van a fundir se puede utilizar un máximo de 15 colores, las imágenes que tengan más deberán ser reducidas a este valor. Los colores deberán estar situados en cualquier caso en la parte inferior de la paleta (*0 hasta número de colores -1*). El número de colores para la imagen estática se calcula mediante la siguiente fórmula:

```
Colores estáticos = 256 - colores dinámicos * (colores dinámicos +1)
```

Los colores estáticos deben colocarse, en cualquier caso, en el extremo superior de la paleta, ya que esta parte es la única que se mantiene sin modificación durante el fundido. Si se desea, por lo tanto, fundir de forma encadenada dos imágenes de 15 colores, restarán *256-15*16= 16 colores* para las imágenes estáticas al final de la

paleta (*desde el color 240 al 255*), en un fundido de 13 colores restarán *256-13*14= 74 colores*, o sea los colores *182 a 255*.

La alternativa rápida - Animación mediante rotación de paletas

Tal como hemos visto hasta ahora es posible cambiar grandes partes de la imagen de una sola vez mediante simples modificaciones de las paletas. Esta gran ventaja de los modos gráficos basados en paletas también se puede utilizar de forma extraordinaria para animaciones reales, en las que no solamente se funden zonas, sino se generan movimientos reales en la pantalla.

¿Cómo es posible conseguirlo? Tomemos como ejemplo un simple punto rojo que se desea desplazar de izquierda a derecha en 10 píxeles. Para ello se dibujan simplemente 10 puntos uno al lado de otro con diferentes colores correlativos en la paleta, en nuestro ejemplo los colores 0 a 9. Si se coloca ahora el color 0 en rojo y los demás nueve colores en negro, solamente estará visible el primer pixel de los diez, teniendo los demás el mismo color que el fondo. Si ahora se desplaza la paleta un punto hacia arriba, de forma que el color 1 sea rojo y el color 0 y los colores 2 a 9 sean negros, se verá el segundo pixel, desapareciendo el primero. Si se sigue desplazando la paleta, el punto se irá moviendo por la pantalla de izquierda a derecha, sin que para ello (después de inicializar los 10 puntos) haya que modificar ni un solo byte de la memoria de vídeo. Cuando el registro rojo llega al borde superior (color 9), se puede volver a enlazar con el color 0, de forma que se produzca un animación cíclica, que mueve el punto de izquierda a derecha y luego vuelve a saltar a la izquierda. También sería posible, por ejemplo, invertir la dirección del movimiento al llegar al final de la paleta, de manera que el punto se mueva de un lado a otro.

Para un único punto este procedimiento tendría poco sentido, ya que en vez de modificar 2 bytes en la memoria de vídeo (borrar punto viejo, colocar el nuevo) deberán colocarse de una vez diez valores de color, y este a través de los lentos accesos a puertos. Esta rotación de la paleta es mucho más efectiva cuando se trata de superficies grandes y/o complicadas. Tanto una gran cantidad de puntos como un compleja descripción matemática de los puntos que se desean mover justifican resumir los movimientos en una rotación de la paleta. Cuando se deseen mover grandes superficies, estas deberán estructurarse de la forma correspondiente. Estas áreas lógicamente no deberán estar compuestas por solamente un color de la paleta, si no que deberán contener un cambio de color en la dirección del movimiento, de forma que se pueda mover a nivel de líneas. Gráficamente se podría explicar con la siguiente figura:

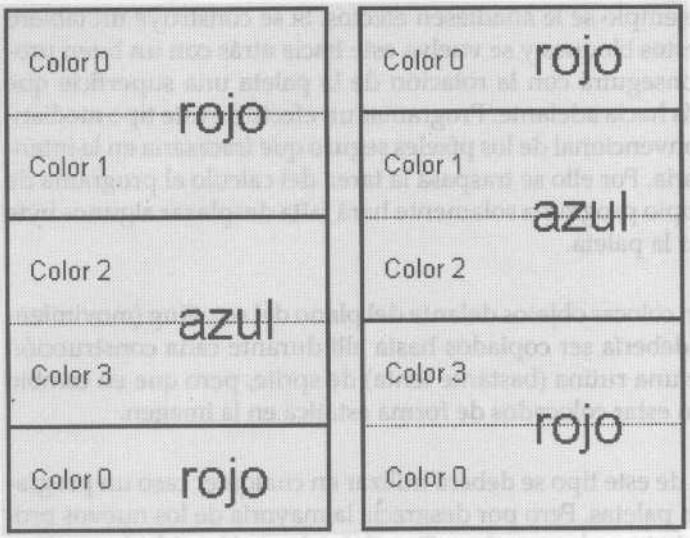

Figura 8: Estructura de los colores para el movimiento

Aquí se efectúa un scroll hacia arriba de bloques de una altura de 2 píxeles, funcionando este método de la misma forma para bloques de altura mayor y de estructuras más complejas. La imagen de ejemplo contienen dos bloques y medio, que están coloreados de forma alternativa en rojo y azul. El número de bloques no tiene ninguna importancia, por lo que la imagen puede ser ampliada sin ningún límite hacia arriba y hacia abajo, siempre que se mantenga el orden de los colores; también son posibles bloques parciales.

La generación de la imagen total es periódica, lo que significa que después de dos bloques se vuelve a repetir la construcción, por lo que dos bloques están rellenados con un recorrido de colores (1) relacionado desde el color 0 al color 3. A continuación se sitúan en la paleta los colores 0 y 1 en rojo y los colores 2 y 3 en azul, de forma que los bloques aparezcan de esta forma. Las preparaciones se realizan de la mejor forma con un programa de dibujo, de forma que nuestro programa ya solamente se tenga que encargar del movimiento.

El movimiento se genera desplazando hacia abajo la paleta(parcial) desde el color 0 al color 3. El color 0 contendrá ahora el color 1, el color 1 del 2, el 2 el color del 3 y el 3 el color del color 0, de forma que se produce una rotación. Tal como muestra la imagen, las zonas azules y rojas se desplazan una línea hacia arriba, lo cual era justo lo que se pretendía.

Pero este scrolling tampoco tiene mucho sentido, ya que se podría conseguir lo mismo modificando la dirección de inicio de la pantalla (*Linear Starting Address*). Este procedimiento se volvería interesante en el momento en el que a la simple

estructura del bloque ejemplo se le añadiesen efectos. Si se construye un tablero de ajedrez a partir de estos bloques y se vuelva este hacia atrás con un buen programa de dibujo, se conseguirá con la rotación de la paleta una superficie que efectúa un scroll de atrás hacia adelante. Programar un efecto de este tipo mediante una manipulación convencional de los píxeles seguro que fracasaría en la intensidad de calculo necesaria. Por ello se traspasa la tarea del calculo al programa de dibujo, y dentro del propio programa solamente hará falta desplazar algunos byte en la memoria y colocar la paleta.

Otra ventaja consiste en colocar objetos delante del plano del scrolling (movimiento), que normalmente debería ser copiados hasta allí durante cada construcción de la imagen mediante una rutina (bastante lenta) de sprite, pero que en cambio con esta técnica pueden estar colocados de forma estática en la imagen.

Al generar una imagen de este tipo se deberá utilizar en cualquier caso un programa de dibujo basado en paletas. Pero por desgracia la mayoría de los nuevos programas de Windows trabajan sobre una base TrueColor, lo cual impide determinar qué color de la paleta tendrá más adelante un punto. Para generar degradados de colores relacionados se necesita un control directo de la paleta.

La imagen de ejemplo SCHACH.GIF ha sido creada mediante el programa Deluxe Paint II Enhanced, el cual domina tanto las perspectivas como las manipulaciones directas de paletas. Primero se dibujó un tablero de ajedrez en blanco y negro y se rellenaron todos los bloques blancos con los colores 16 a 31 (degradado de colores de rojo a azul) y todos los negros con los colores 32 a 47. Ahora este tablero de ajedrez puede ser manipulado libremente, ya que la apariencia gráfica no influye en la rotación de la paleta; en nuestro ejemplo lo hemos volcado hacia atrás, de forma que represente un plano.

Por encima del tablero de ajedrez se ha colocado un logotipo, que ocupa la parte sin utilizar del borde superior de la paleta. A la derecha y por encima del tablero se ve otra aplicación un poco diferente de la rotación de paleta, que se utiliza muchas veces en aventuras gráficas: una fuente. Esta fuente utiliza los colores 48 a 63, que contienen un degradado desde el negro hasta el azul, el cual delante de un fondo negro provoca el efecto de agua en movimiento.

Como último efecto la imagen contiene un "radar" rojo. Aquí simplemente se ha generado un degradado circular de los colores 64 a 88. Este degradado de colores se compone de 25 círculos concéntricos que poseen valores de colores seguidos. En la paleta solamente están colocados en rojo los colores 64 hasta 80, el resto está en negro. Durante una rotación los dos aros de color rojo se mueven por encima de los aros con los colores 64 a 80 y generan de esta forma círculos concéntricos que se separan.

Para controlar de forma simple los efectos dibujados el programa Deluxe Paint posee una función llamada *farbrolle,* la cual realiza la rotación (muy lenta). Pero ¿cómo se puede generar este efecto en sus propios programas? Para ello sirve el procedimiento *pal_rot,* que se encuentra en *ModeXlib.asm,* a la cual se le traspasa como parámetro la zona que se desea rotar. Normalmente la zona descrita se desplaza en una posición hacia abajo, hacia el número de color más bajo, y se vuelve a añadir el color inferior en la parte superior. En cuanto el primer límite pasado es superior al segundo, se rota hacia arriba:

```
Pal_Rot proc pascal far Start,Ende:Word
;rota parte de paleta Start a Ende en 1
;si Start < Ende : rotación hacia abajo
;si Start > Ende : rotación hacia arriba

    mov ax,ds                   ;es al segmento de datos
    mov es,ax

    lea si,palette              ;cargar offset de paleta
    mov di,si                   ;también a di

    mov ax,3                    ;convertir «Start» en offset de paleta
    mul start
    add si,ax                   ;y sumar a si
    mov ax,3                    ;lo mismo para destino
    mul ende
    add di,ax                   ;sumar a di

    mov bx,[si]                 ;guardar bytes de color inicial
    mov dl,[si+2]

    mov cx,di                   ;diferencia entre Start y Ende es nº
    sub cx,si                   ;de bytes a copiar

    mov di,si                   ;color inicial como offset de destino
    add si,3                    ;un color por encima como offset fuente
                                ;listo para copiar adelante
    cld                         ;predeterminado: copiar adelante
    or cx,cx                    ;si cx negativo (Start > Ende)
    jns adelante
    std                         ;copiar hacia atrás
    neg cx                      ;corregir cx
    sub si,4                    ;si al 2. Byte del penúltimo color
    add di,2                    ;di al 2. Byte del último color
    add cx,2                    ;copiar 2 bytes más,
adelante:                       ;que posición del bucle dé copia correcta

    rep movsb                   ;copiar colores
    mov [di],bx                 ;bytes del antiguo color inicial
    mov [di+2],dl               ;escribir como último color
```

```
cld                    ;borrar Direction-Flag
ret
Pal_Rot Endp
```

Después de haber convertido los colores recibidos en offsets de byte, se guarda primero el color inicial, ya que los demás colores serán desplazados en dirección a este color inicial, por lo cual lo borrarán. La cantidad de bytes a desplazar se calcula simplemente mediante la diferencia de los dos offsets, que se evalúa a continuación. Para preparar el desplazamiento en sí, se coloca *DI* en el offset de inicio, ya que se va a desplazar hacia ahí. *SI* se coloca primero en el siguiente color, ya que se va a desplazar hacia adelante, de forma que ésta se copia en el primer paso al color de inicio.

Cuando el valor de inicio es superior al valor final (se ve en un número negativo de bytes a copiar), habrá que adaptar algunos registros al desplazamiento hacia arriba. Para ello habrá que colocar primero el flag de dirección (*direction flag*), ya que en un desplazamiento de memoria de abajo hacia arriba hay que copiar al revés, ya que en caso contrario se borrarán con los valores copiados los valores aún no copiados, lo cual sería un algoritmo bastante poco efectivo para llenar una zona de la memoria. Dado que *CX* es negativo, aún se deberá cambiar el prefijo.

A continuación habrá que igualar *DI y SI*, ya que *SI* se encuentra hasta este momento en el valor de color que sigue inmediatamente en la paleta (antes se desplazó ahí mediante *add si, 3*, lo cual es correcto en un movimiento hacia delante). Por esta razón *SI* deberá ser colocado en el último byte del penúltimo color (el último byte, ya que se copia mediante *rep movsb!*). *DI* también deberá indicar al último (segundo) byte del último color, a fin de rellenar este color y los situados por debajo desde arriba hacia abajo.

Debido al posicionamiento, *DI* no apuntará después de la copia al color inferior. Esto se podría igualar incluyendo después del bucle una segunda pregunta a la dirección de copiado, y corrigiendo *DI* de forma correspondiente. Pero es mucho más simple copiar dos bytes más, que serán sustituidos después del bucle de copiado por los dos bytes guardados del color inicial. Este bucle y la sustitución del bucle de copiado son iguales para ambas direcciones y siguen por lo tanto directamente después de la etiqueta *adelante*. Finalmente se borra el flag de dirección (*Direction Flag*), a fin de evitar en otros procedimientos cualquier caos a causa de la dirección invertida de la cadena. Para la imagen de ejemplo *schach.gif* hemos añadido en el CD el correspondiente programa de ejemplo, llamado *palrot.pas*:

```
Uses Crt,ModeXLib,Gif;

Var slow_flag:Boolean;          {para controlar los degradados lentos}

Begin
  Init_Mode13;                  {activar modo 13h}
  LoadGif('schach');            {cargar y mostrar imagen}
  Show_Pic13;
  Repeat
    Pal_Rot(16,47);             {mover "tablero de ajedrez"}
    If slow_flag Then Begin     {en cada segunda pasada:}
      Pal_Rot(63,48);           {mover "fuente"}
      Pal_Rot(88,64);           {mover "radar"}
    End;
    slow_flag:=not slow_flag;   {"fuente" y "radar"}
                                {permitir y bloquear alternativamente}
    WaitRetrace;                {sincronización}

    SetPal;                     {fijar la paleta rotada}
  Until KeyPressed;             {hasta tecla}
  TextMode(3);
End.
```

Aquí se carga simplemente la imagen en la pantalla en el modo *13h* y se rotan dentro del bucle las zonas de la paleta con los efectos individuales en la dirección correspondiente. No estaría de más experimentar un poco con las direcciones, cambiando los valores que se traspasan a *Pal_rot*.

Mientras que el tablero de ajedrez deberá desplazarse a toda velocidad, esto es demasiado rápido para la fuente y el radar. Por ello sus partes de la paleta solamente se rotarán en cada segunda pasada por el bucle (controlado por la variable booleana *slow_flag*, que va cambiando entre *TRUE y FALSE*).

6.11 El monitor en llamas - Fuego

Ver un fuego en la pantalla ya no es nada nuevo después de las grandes cantidades de juegos de rol que han salido al mercado. Pero estas llamas se basan en dibujos hechos manualmente que son reproducidos uno detrás de otro. Aquí vamos a explicar otra posibilidad, la cual se utiliza en muchas demos: imitar directamente la estructura de las llamas. Para ello hay que tener en cuenta dos puntos básicos: un fuego se mueve constantemente hacia arriba, y además en su parte inferior es blanco y se diluye hacia arriba en tonos rojos. El programa *FLAMES.PAS* muestra de forma espectacular este procedimiento. Se necesitan dos búferes a fin de reflejar la memoria de vídeo (mejor dicho su parte inferior) en la rápida RAM del sistema:

```
Uses Crt,ModeXLib;
Type Block=Array[0..99,0..319] of Byte;
Var
  Src_Frame,                    {imagen anterior}
  Dest_Frame:^Block;            {imagen actual}
```

Las rutinas ShowScreen y Prep_Pal así como el programa principal no debrían plantear ninguna dificultad de comprensión:

```
Procedure Show_Screen;          {copia pantalla acabada en VGA}
Var temp:Pointer;               {para permutar los punteros}
Begin
asm
  push ds
  lds si,Dest_Frame             {imagen acabada como fuente}
  mov ax,0a000h                 {VGA como destino}
  mov es,ax
  mov di,320*100                {a partir de línea 100}
  mov cx,320*100/4              {copiar 100 líneas como DWords}
db 66h                          {Operand Size Prefix (32 Bit)}
  rep movsw                     {copiar}
  pop ds
End;
  temp:=Dest_Frame;             {permutar ptro. fuente y destino}
  Dest_Frame:=Src_Frame;
  Src_Frame:=temp;
End;

Procedure Prep_Pal;             {preparar paleta a Flames}
Var i:Word;
Begin
  FillChar(Palette,80*3,0);     {base: todo negro}
  For i:=0 to 7 do Begin
    Palette[i*3+2]:=i*2;        {color 0-7: sube azul}
    Palette[(i+8)*3+2]:=16-i*2; {color 0-7: baja azul}
  End;
  For i:=8 to 31 do             {color 8 -31: sube rojo}
    Palette[i*3]:=(i-8)*63 div 23;
  For i:=32 to 55 do Begin      {color 32-55: sube verde, rojo igual}
    Palette[i*3]:=63;
    Palette[i*3+1]:=(i-32)*63 div 23;
  End;
  For i:=56 to 79 do Begin      {color 56-79: sólo sube azul}
    Palette[i*3]:=63;
    Palette[i*3+1]:=63;
    Palette[i*3+2]:=(i-56)*63 div 23;
  End;
  FillChar(Palette[80*3],176*3,63);  {resto blanco}
  SetPal;                       {fijar paleta acabada}
End;
```

```
begin
  Randomize;                        {determinar Random Seed}
  GetMem(Src_Frame,320*100);        {obtener memoria para imagen fuente}
  FillChar(Src_Frame^,320*100,0);
  GetMem(Dest_Frame,320*100);       {obtener memoria para imagen destino}
  FillChar(Dest_Frame^,320*100,0);
  Init_Mode13;                      {activar modo 13h}
  Prep_Pal;                         {preparar paleta}
  Repeat
    Scroll_Up;                      {llamas hacia arriba}
    New_Line;                       {añadir nueva línea abajo}
    Show_Screen;                    {mostrar pantalla acabada}
  Until KeyPressed;
  TextMode(3);
end.
```

El programa principal primero direcciona y borra ambos búferes, activa el modo *13h* y prepara la paleta para las llamas. El resultado queda explicado por el siguiente gráfico:

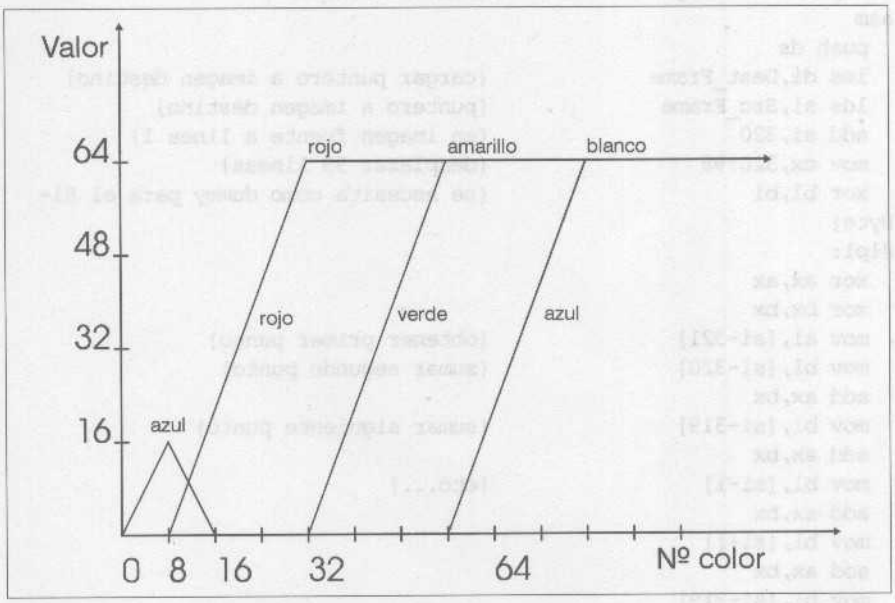

Figura 9: Paleta de colores para las llamas

Los píxeles con números de color más bajos representarán más adelante las zonas más frías de las llamas, situados aquí en la zona izquierda. Si se lee el gráfico de derecha a izquierda , o sea desde las altas a las bajas temperaturas, se verá lo siguiente:

Primero todos los componentes de color están al máximo (=blanco), a partir del color 80 se va reduciendo progresivamente el componente azul, de forma que los colores van pareciendo más amarillentos. A partir del color 56 se va reduciendo además el componente verde, de forma que al final solo queda el rojo. Después de fundir también este color (a partir del número 32), el resultado normal sería negro. Pero en este caso se le añade además un pequeño brillo azul (colores 16 a 0) que se sitúa por encima de las llamas. Al que no le guste, simplemente deberá eliminar el primer bucle del procedimiento.

El bucle principal se compone de tres instrucciones: *Scroll_Up, New_line y Show_screen*. Esto quiere decir que después de preparar la nueva imagen en el búfer *Dest_Frame*, se lleva a la pantalla mediante *Show_Screen*. Para ello se utiliza un simple bucle de copiado (*Dword*). A continuación se invierten los punteros a los dos búferes, de forma que la pantalla ya generada se utilizará en la siguiente pasada como fuente. Los procedimientos centrales son sin lugar a dudas *Scroll_Up y New_line:*

```
Procedure Scroll_Up;assembler;
{desplaza la imagen una línea arriba e interpola}
asm
  push ds
  les di,Dest_Frame              {cargar puntero a imagen destino}
  lds si,Src_Frame               {puntero a imagen destino}
  add si,320                     {en imagen fuente a línea 1}
  mov cx,320*98                  {desplazar 99 líneas}
  xor bl,bl                      {se necesita como dummy para el Hi-
Byte}
@lp1:
  xor ax,ax
  xor bx,bx
  mov al,[si-321]                {obtener primer punto}
  mov bl,[si-320]                {sumar segundo punto}
  add ax,bx
  mov bl,[si-319]                {sumar siguiente punto}
  add ax,bx
  mov bl,[si-1]                  {etc...}
  add ax,bx
  mov bl,[si+1]
  add ax,bx
  mov bl,[si+319]
  add ax,bx
  mov bl,[si+320]
  adc ax,bx
  mov bl,[si+321]
  adc ax,bx
  shr ax,3

  or ax,ax                       {¿ya es 0?}
  je @null
```

```
    dec al                      {si no, decrementar}
  @null:
    stosb                       {valor a destino}
    inc si                      {siguiente punto}
    dec cx                      {otros puntos?}
    jne @lp1
    pop ds
  End;

  Procedure New_Line;           {reconstruye las líneas inferiores}
  Var i,x:Word;
  Begin
    For x:=0 to 319 do Begin    {3 líneas inferiores con nºs aleat.}
      Dest_Frame^[97,x]:=Random(15)+64;
      Dest_Frame^[98,x]:=Random(15)+64;
      Dest_Frame^[99,x]:=Random(15)+64;
    End;
    For i:=0 to Random(45) do Begin {insertar nº aleat. de hotspots}
      x:=Random(320);           {en coord. casuales}
      asm
        les di,Dest_Frame       {direccionar imagen destino}
        add di,98*320           {procesar línea 98 (penúltima)}
        add di,x                {sumar coord. x}
        mov al,0ffh             {color más claro}
        mov es:[di-321],al      {crear Hotspot grande (9 puntos)}
        mov es:[di-320],al
        mov es:[di-319],al
        mov es:[di-1],al
        mov es:[di],al
        mov es:[di+1],al
        mov es:[di+319],al
        mov es:[di+320],al
        mov es:[di+321],al
      End;
    End;
  End;
```

El procedimiento *Scroll_up* mueve el mar de llamas una posición hacia arriba, interpolándose al mismo tiempo, lo que significa que cada punto se calcula a partir del valor medio de su entorno. Así pues, de esta forma se consiguen las llamas difuminadas.

Dicho procedimiento extrae en el bucle *@lp1* los 8 puntos limítrofes al punto origen, los suma y divide la suma entre ocho. Esta media se reduce en 1, a fin de evitar que las llamas suban demasiado, y, una vez hecho esto, se escribe al búfer destino.

New_line se encarga ahora de que en el borde inferior de la pantalla se vayan añadiendo líneas nuevas, como si fueran las brasas. En el primer bucle se van llenando las últimas tres líneas con valores aleatorios entre 64 y 79. A continuación se añaden en el segundo bucle los llamados hotspots. Aquí se trata de zonas muy calientes, desde las cuales parten las llamas. Por esta razón se colocan aquí bloques de nueve píxeles en 255, de forma que en el scrolling hacia arriba se enfrían muy poco a poco, y se desplazan de esta forma como zonas brillantes hacia arriba.

6.12 El secreto de Comanche (Tm) - espaciado Voxel

Es una parte de programa utilizada a menudo en las demos y ya se utiliza en algunos juegos: el *Voxel Spacing*. Se trata de un paisaje imaginario compuesto por montañas y valles, por encima del cual se mueve el espectador.

Existen infinidad de métodos para representar esta descripción tan genérica. El número de algoritmos es, por lo menos, igual de grande que el número de programadores que lo han intentado. A continuación vamos a explicar un procedimiento que funciona sin necesidad de complejas imágenes tridimensionales y que en cambio se reduce a relaciones en dos dimensiones.

En principio lo que se hace es volcar un mapa hacia el plano, de forma que el observador obtenga un punto de vista desde arriba y en diagonal. La información de altura solamente se puede representar mediante largas líneas verticales. Como mapa que incluya los altos y bajos del paisaje sirve cualquier simple imagen de 320x200 con colores relacionados. El color 0 corresponde al punto más bajo y el 255 al más alto.

La proyección se consigue en la práctica de forma bidimensional según un principio relativamente simple: los objetos situados más lejos suelen aparecer más pequeños que los que están cerca. Si ahora se mira el paisaje a través de un marco -que no es otra cosa que la pantalla-, se comprobará que en la lejanía caben más objetos dentro del marco que cerca, ya que son más pequeños. Por lo tanto siempre se ve una parte.

El marco rectangular representa todo el paisaje existente desde una perspectiva aérea y el trapecio la parte visible de éste. Ahora solamente hace falta que este trapecio se represente en una zona rectangular de la imagen; para ello se estira la parte delantera de forma horizontal, de forma que los objetos aparezcan aquí ampliados. Las coordenadas indicadas hace referencia a una proyección hacia la parte inferior de la pantalla del modo X.

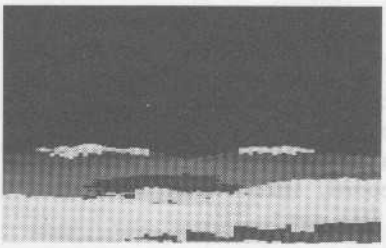

Figura 10: Paisaje dibujado con gráficos Espacio Voxel

La proyección en forma de trapecio se consigue reduciendo simplemente el número de píxeles que caben en una línea. Expresado de otra forma: la proporción entre el paisaje y el tamaño de los píxeles de pantalla se va reduciendo. El valor original se de 1:1, o sea que caben 80 píxeles en una línea (por razones de velocidad se utiliza esta resolución), reduciéndose durante la creación del dibujo esta proporción, hasta conseguirse una relación de 1:2, de forma que quepan solamente 40 píxeles en la línea del primer plano.

En la representación por líneas se deberá tener en cuenta que las separaciones entre las líneas se vayan estrechando hacia atrás, o sea que si se dibuja de arriba hacia abajo deberá incrementarse progresivamente la separación entre líneas.

Ahora solamente falta la información sobre la altura en la imagen. Para ello se utiliza, como hemos dicho antes, el color de cada punto, ordenándose un número de píxeles uno encima de otro. Cuanto mayor sea el valor del color, más se convertirá el pixel en una línea vertical. Mediante esta estructura vertical se evitan además espacios en blanco que se producen a causa de la creciente separación entre líneas.

Para representar todo ello de la forma más real posible aún faltan las superficies de agua. Para ello se coloca simplemente cada color situado por debajo de un valor determinado en justo ese valor. De esta forma se crean planos que no poseen ninguna estructura vertical. Para generar los paisajes lo mejor es utilizar un generador fractal, el cual puede generar nubes de plasma, siendo muy indicado el programa *FractInt*. Genere simplemente un campo de plasma en una resolución de 320 x 200 y cargue a continuación (mediante [C]+[L]) la paleta LANDSCAP.MAP. La imagen final se guarda mediante [ESC] + [S].

El programa *VOXEL.PAS* redibuja mediante un bucle la escena cada vez que se mueve el ratón (control a través del ratón).

```
Uses Crt,Gif,ModeXLib;
Var x,y:Integer;                {coordenadas del trapecio}

Procedure Draw_Voxel;external;
```

```
{$l voxel.obj}

Begin
  asm mov ax,0; int 33h End; {reponer controlador de ratón}
  Init_ModeX;                {activar modo X}
  LoadGif('landsc3');        {cargar paisaje}
  x:=195;                    {determinar coord. inicial}
  y:=130;
  Repeat
    ClrX($0f);               {borrar pantalla}
    Draw_Voxel;              {dibujar paisaje}
    Switch;                  {activar pág. de pantalla acabada}
    WaitRetrace;             {esperar retrazado}
    asm
      mov ax,000bh           {función 0bh: leer coord. relativas}
      int 33h
      sar cx,2               {división por 2}
      sar dx,2
      add x,cx
      add y,dx
    End;
    If x < 0 Then x:=0; If x > 130 Then x:=130;
    If y < 0 Then y:=0; If y > 130 Then y:=130;
  Until KeyPressed;          {hasta tecla}
  TextMode(3);
End.
```

El procedimiento central se llama *Draw_Voxel*, el cual se encuentra en el módulo en ensamblador *Voxel.asm*:

```
data segment
  extrn vscreen:dword        ;puntero a datos de paisaje
  extrn x,y: word            ;coordenadas del trapecio
  extrn vpage:word           ;pág. pantalla actual
data ends

code segment
assume cs:code,ds:data

;variables con decimales (8 bits bajos):

offst dd 0                   ;offset actual
step dd 0                    ;tamaño de píxel
row_start dd 0               ;comienzo de la línea actual
row_step dd 0                ;distancia a la sig. línea


z_count dw 0                 ;contador de profundidad
shrink dw 0                  ;correcc. en el borde inferior
```

```
Linea dd 0                  ;nº de línea de pant. actual
vpage_cs dw 0               ;página de pant. en el segm. de código


.386
public Draw_Voxel
Draw_Voxel proc pascal
;representa paisaje en la pág. de pantalla actual
;lee datos de vscreen a partir de posición (x/y)

  mov ax,vpage              ;guardar nº de pág. de pantalla
  mov vpage_cs,ax

  push ds
  mov ax,0a000h             ;cargar segm. destino
  mov es,ax

  mov ax,320                ;calcular offset de paisaje
  imul y
  add ax,x

  lds si,vscreen            ;tomar datos de vscreen
  add si,ax                 ;sumar offset
  shl esi,8                 ;convertir en nº de coma fija
  mov offst,esi             ;valores iniciales para píxel...
  mov row_start,esi         ;... y línea

  mov step,100h             ;primero factor de escalado 1
  mov Linea,100*256         ;comenzar en línea de pant. 100
  mov row_step,14040h       ;distancia de líneas 320,25
  mov shrink,0              ;de momento no hay corrección
  mov z_count,160           ;nº de línea a calcular
```

La primera parte inicializa algunas variables importantes, como son *Offst y row_start*. Ambas forman un valor de coma fija (parte anterior a la coma en los bits 8-31, parte posterior en los bit 0-7), el cual indica el offset del pixel y de la línea actual respectivamente. Después de cada punto se incrementa *Offst* en un *Step* con lo cual se avanza un paso dentro del paisaje. Si *Step* es, por ejemplo, *080h* (= 0,5), cada dos pasos se leerá un pixel nuevo del paisaje.

Algo similar ocurre con *row_start y row_step*: *row_step* contiene 14040, lo cual se corresponde en modo decimal al valor de 320.25. Gracias a ello se coloca después de cada línea el *row_start* al principio de la siguiente línea, y vuelve a comenzar desde el principio el estrechamiento, comenzando la siguiente línea un poco más hacia la derecha.

```
next_y:
  mov eax,Linea              ;obtener nº de línea (pantalla) actual
  mov ebx,eax                ;guardar
  shr eax,8                  ;convertir en nº entero
  add eax,50                 ;50 Pixels hacia abajo
  imul eax,80                ;convertir en offset
  mov di,ax                  ;guardar como ptro. destino

  cmp di,199*80              ;¿sobrepasado borde pantalla?
  jb normal
  mov di,199*80              ;sí, posicionar a última línea

  mov eax,Linea              ;diferencia al borde inferior
  shr eax,8
  sub eax,149
  mov shrink,ax              ;y guardar como corrección

normal:
  add di,vpage_cs            ;sumar página de pantalla actual

  imul ebx,16500             ;multiplicar con 1.007
  shr ebx,14                 ;sumar * 16500 / 16384
  mov Linea,ebx              ;y guardar
```

Aquí se calcula el offset de cada línea. La variable *Linea* contiene la línea actual en la pantalla, que aquí se convierte en un offset. El número de línea no se debe confundir con el *row_Offst*; esta variable se encarga de la posición dentro del paisaje, mientras que *Linea* indica la posición en pantalla. Cuando el offset calculado está por debajo de la pantalla, se deberá dibujar a pesar de todo, ya que puede ser que las líneas verticales de ese punto entren en la pantalla. Ahora se analiza este caso especial, colocando primero el puntero destino *di* en la pantalla y reduciendo luego la altura de la barra (por el numero de líneas indicado en *Shrink*).

En este momento entra en juego la separación creciente hacia adelante de las líneas de pantalla. La coordenada Y actual de la pantalla se encuentra en la variable *Linea*, que aquí se multiplica por el factor 1,007, de forma que las líneas se van separando conforme se avanza.

```
  mov bp,80                  ;nº píxeles por línea
next_x:
  mov esi,offst              ;cargar offset de píxel actual
  shr esi,8                  ;convertir en nº entero
  xor eax,eax
  mov al,[si]                ;cargar punto del paisaje
  mov cx,ax                  ;guardar

  cmp cx,99                  ;color (=altura) < 100
```

```
    ja fill_bar
    mov ax,99                   ;poner a 99

  fill_bar:
    shl ax,5                    ;proyección con pto fuga: =altura * 32
    xor dx,dx
    push bp
    mov bp,z_count              ;dividido por distancia
    add bp,50
    idiv bp
    pop bp

    sub ax,shrink               ;realizar corrección
    jbe seguir                  ;si <= 0, no dibujar
```

Para cada punto se lee ahora el color en el offset actual y se establece el nivel del mar mediante un valor mínimo de 99 . Mediante el contador *z_count*, que indica prácticamente la distancia de cada línea hasta el observador, se calcula la perspectiva del punto de fuga. Este procedimiento se explicará más en detalle en el apartado 8.2. Ahora se reduce la altura (guardada en *al)* por el valor de *Shrink*, a fin de poder trabajar con la parte inferior de la pantalla.

```
    push di
  next_fill:
    mov es:[di],cl              ;guardar color
    sub di,80                   ;direccionar siguiente línea
    dec al                      ;decrementar contador
    jne next_fill               ;¿seguir?
    pop di

  seguir:
    inc di                      ;direccionar siguiente byte en pantalla
    mov esi,step                ;obtener amplitud de paso
    add esi,offst               ;sumar
    mov offst,esi               ;y escribir

    dec bp                      ;siguiente punto
    jne next_x

    mov esi,row_step            ;desplazar inicio de línea
    add esi,row_start
    mov row_start,esi
    mov offst,esi               ;cargar offset de píxeles de nuevo

    dec step                    ;decrementar factor de escalado
    dec z_count                 ;seguir contador de líneas
    jne next_y
    pop ds
    ret
```

```
Draw_Voxel endp

code ends
end
```

En el bucle *next_fill* se dibuja ahora la barra vertical en el tamaño indicado. A continuación se calcula el offset del siguiente punto, y en caso de que comience una nueva línea, también el suyo. Además se reduce la anchura de paso de *Step* después de cada línea, a fin de conseguir el ensanchado hacia adelante.

7. Sprites - acción rasante en la pantalla

Ya sea como nave espacial en un juego de acción o como figura en un arcade, en cualquier lado nos encontramos hoy en día con los sprites. Cualquier cosa que se mueva en pantalla de forma diferente al fondo se puede considerar en principio un sprite, o sea, que todo el mundo sabe en el fondo lo que es. Pero nos queda una pregunta: ¿cómo se programa esto?

7.1 Los fundamentos

Para poder generar sprites uno mismo habrá que pensar primero en su estructura interna. Los sprites no son el fondo nada más que pequeñas (o grandes, según la necesidad de cálculo que precisen) partes de un gráfico que pueden ser ubicadas libremente sobre una imagen de fondo.

Pero tienen que cumplir por lo menos una condición más: deben existir partes transparentes a través de las cuales se ve el fondo. Esto no es solamente necesario en los "agujeros" en medio de los sprites, sino que su aplicación más común suelen ser los bordes de los mismos. Cada sprite que no tenga una estructura rectangular al 100% deberá tener en el borde zonas transparentes. Al efectuar la representación no se pueden tener en cuenta formas circulares o similares a causa de razones de velocidad, de forma que fundamentalmente se representan en pantalla formas rectangulares, por lo que los bordes deben ser rellenados, ya que, en caso contrario y con independencia de su forma, siempre aparecerían rectángulos rellenos.

Mientras que los ordenadores domésticos se encargan de casi todo el trabajo, y los programadores ya solamente tienen que calcular la posición, esta tarea se complica bastante más en los PCs. Todo el proceso se lleva a cabo en la memoria de vídeo de la VGA, por lo que se deberá trabajar de una forma optimizada al máximo. Pero la solución en base al software en los PCs también tiene ventajas incalculables: la generación está completamente en manos del programador, en teoría es posible trabajar con cualquier tamaño de sprite, aunque en algún momento se pueda superar la capacidad de cálculo del ordenador. También son posibles escalados y rotaciones, o sea que somos libres de crear a nuestro antojo - siempre que sepamos hacerlo-. Hay que tener en cuenta los procedimientos fundamentales para la representación de sprites:

1. Borrado de los sprites mediante copiado del fondo de la imagen a la página de pantalla actual.

2. Movimiento y representación de los sprites.
3. Cambio a la página preparada.

Antes de poder representar los sprites en su nueva posición, primero deberán desaparecer todos de la pantalla. Copiar para ello toda una página de pantalla puede parecer superfluo. ¿No sería posible guardar el fondo antes de la representación de cada sprite y a continuación volverlo a escribir? Este procedimiento solamente tendría sentido con uno o máximo dos sprites, ya que el trabajo de administración es enorme.

Por un lado, solamente sería posible guardar un fondo en la RAM de la VGA, dado que aquí existe un acceso rápido. Pero esta RAM es muy limitada, y llegaría un momento en que no quedaría espacio. Por otro lado, el trabajo de administración es superior al de copiar una página de pantalla, ya que en definitiva se trata de partes situadas en medio de la pantalla, que solamente pueden ser copiadas por líneas, ya que un *rep movsb* no es posible aplicarlo a toda la pantalla.

O sea, que cada vez se copia toda una página de fondo, que se sitúa lógicamente en la RAM de la VGA, por ejemplo en la página 3, y que se traslada después mediante un rápido bucle de copiado a la página actual de una sola vez.

En el movimiento de los sprites existe una libertad total, se pueden mover en forma lineal por la pantalla, describir círculos, moverse por una línea senoidal o ser controlados por música. Cada programador es muy suyo a la hora de decidir qué hacer con ellos.

Antes de describir la representación de sprites vamos a aclarar el concepto de dos páginas de pantalla: como ya hemos explicado anteriormente, es casi imposible efectuar grandes manipulaciones de la imagen durante un retrace vertical, pero, por otro lado, esta es la única posibilidad que existe para evitar centelleos y deformaciones de los sprites. Además, en el procedimiento explicado anteriormente (entre los pasos 2 y 3) temporalmente no existe ningún sprite en la pantalla, lo cual se ve claramente en el centelleo de todos los sprites.

Por ello habrá que utilizar la segunda página de pantalla, a fin de preparar esta página invisible, mientras que la visible permanece invariable. Después de efectuar todas las modificaciones se cambia a la nueva página de pantalla, y se genera una nueva utilizando la que estaba visible anteriormente.

Para la representación en sí existen dos conceptos fundamentales: por un lado se pueden mantener los datos de los sprites en la memoria de vídeo, por ejemplo, en la página 2, y copiarlos desde ahí, cuando sea necesario, a la página actual. Esto tiene dos desventajas fundamentales: el rápido modo de escritura *1* se basa en bloques enteros de cuatro (un byte de cada plano), de forma que es imposible

copiar, por ejemplo, un sprite desde la coordenada X 5 (Plano *1*) a la 6 (plano 2), ya que los datos solamente pueden ser copiados de forma rápida dentro de un mismo plano. La única solución a este problema es mantener cada sprite cuatro veces en la memoria, una vez en la posición X *0*, una en la posición X *1*, etc., lo cual reduce considerablemente el número de sprites disponibles. La segunda desventaja de este concepto radica en el enmascaramiento de los puntos transparentes, lo cual solamente es posible a través de máscaras separadas almacenadas en la memoria principal, que deben ser generadas con anterioridad.

7.2 Leer y escribir sprites

La alternativa es un segundo concepto que también se utiliza en este libro: los datos de los sprites se mantienen en la memoria principal y se llevan a la pantalla punto a punto. La desventaja de este proceso es su reducida velocidad, de forma que habrá que intentar optimizar las rutinas al máximo.

Esto ya se inicia en el formato de los datos de los sprites en la memoria principal. Los datos gráficos se dividen al colocar un sprite entre los cuatro planos, aterrizando los puntos de direcciones divisibles entre cuatro en el plano 0, los divisibles con *resto 1* en el plano 1, etc. La posibilidad más simple para colocar los datos en la pantalla consiste en colocar los puntos en su orden "normal" cambiando constantemente de plano de escritura.

En el procedimiento *P13_2_ModeX* ya vimos que esta no es la solución más rápida. Dicho procedimiento utiliza otro camino, que en principio también puede ser utilizado para los sprites: primero se copian todos los datos de un plano, después se cambia y se copian los siguientes datos.

De ahí proviene el formato de los sprites: los datos no están en formato del *modo 13h* (alineación lineal de los píxeles), sino en un orden similar al modo X. Primero van todos los datos del primer plano, después los del segundo, etc. Al escribir los sprites se podrá acceder sin ningún esfuerzo adicional a los datos, solamente habrá que cargar la dirección de inicio en el registro *SI*, copiar el primer plano, y trabajar a continuación con el siguiente plano, sin que haga falta reposicionar el registro *SI*.

Los datos en sí ya contienen los valores de color, por lo cual pueden ser copiados directamente a la memoria de vídeo. El único color que es especial es el color 0: este color representa el fondo, o sea las partes donde el sprite es transparente. Un problema adicional se plantea en el cálculo de la anchura del sprite, ya que esta no es igual en todos los planos. Imaginémonos, por ejemplo, un sprite de una anchura de 5 píxeles, que debe ser escrito a la posición X 0. En este caso se deberán copiar 2 bytes al plano 0 (píxeles 0 y 4), mientras que los demás solamente contienen un byte.

En los algoritmos que presentamos en este libro este reparto se efectúa de forma global, o sea que no depende de la coordenada X de la posición destino. Si se copia por ejemplo, el sprite a la posición X *1*, el bloque de 2 bytes de tamaño se encontrará en el plano físico *1*, pero dado que el bucle de copiado comienza en este plano, desde el punto de vista de los datos del sprite seguirá tratándose su plano 2, que seguirá teniendo una anchura de 2 bytes. En otras palabras: con la anchura del sprite se puede crear un array con la anchura de los diferentes planos, el cual alimenta la variable contador del bucle de copiado. El primer plano de los datos del sprite de nuestro ejemplo siempre tienen un anchura de 2 byte, y los demás 1 byte, independientemente de la posición actual. Al colocar el sprite solamente habrá que calcular el plano con el cual se comienza para comunicárselo a la VGA.

7.3 Más allá de todas las fronteras - Clipping

Además de tener en cuenta el 0 en los datos de los sprites su colocación plantea otro problema de velocidad: el clipping. No existe casi ningún programa en el que se pueda evitar que un sprite toque el borde de la imagen y desaparezca detrás de este borde. Si se copian los datos según el sistema explicado anteriormente a ciegas a la memoria de vídeo, se producirán unos resultados bastante confusos: al sobrepasar el borde derecho o inferior de la pantalla el resto del sprite se copiará a la siguiente línea o página de pantalla, escribiéndose el resto del sprite en el borde izquierdo o superior de la siguiente línea o página de pantalla. Ninguno de estos casos es deseable, por lo cual se deberá encontrar un algoritmo que controle el clipping, o sea el "recorte" de los datos sobrantes.

La primera posibilidad, y al mismo tiempo la más lenta, consiste en comprobar, antes de colocar cualquier punto, que se esté dentro de los límites de la pantalla. Una opción bastante más lograda es modificar las condiciones del marco antes de entrar en el bucle de copiado, de tal forma que la representación se pueda llevar a cabo de forma convencional. Con ello se intenta mantener el clipping fuera del bucle, ya que cuanto más a fuera esté situada la orden dentro de una estructura anidada de bucles, menos veces se producirá y menos tiempo de cálculo se precisará.

En principio el clipping es bastante simple: se igualan la altura o la anchura y se representa el sprite. En el borde inferior y en el derecho hay que introducir una variable que se encargue de que la zona no representada también sea omitida en los datos de origen. Si una línea de sprite solamente se copia a medias, habrá que encargarse que después de esta línea el puntero a los datos origen salte al principio de la siguiente línea. Esto se debe aplicar de forma paralela al borde inferior, siendo aquí la distancia hasta los datos del siguiente plano.

En los bordes superior e izquierdo aparece un problema adicional: la coordenada en la cual comienza la representación debe ser colocada a 0, a fin de que se dibuje dentro de la pantalla real. Con ello se modifican también el plano inicial y el offset de inicio de la primera columna a copiar en el borde izquierdo. Si un sprite se desplaza por ejemplo un punto más allá del borde izquierdo, la primera columna se encontrará en la coordenada -1, por lo que no deberá ser dibujada. La siguiente columna del primer plano del sprite se encontrará ahora en la coordenada X 3, de forma que se deberá igualar este valor.

En el clipping en el borde derecho e izquierdo hay que tener en cuenta nuevamente el reparto entre los diferentes planos, utilizándose un principio similar al que se ha utilizado para el cálculo de la anchura, pero partiendo ahora a partir de los píxeles sobrantes. Para no complicarlo demasiado partiremos de la base que un clipping en el borde izquierdo de la pantalla anula un clipping en la derecha, ya que para ello se necesitaría un sprite con una anchura de 321 píxeles, algo que en la practica no se suele dar, de forma que podemos saltarnos la combinación borde izquierdo / borde derecho.

7.4 La unidad sprite

Todas estas características especiales se encuentran en los procedimientos *getsprite* y *Putsprite* de la unidad *Sprites:*

```
Unit Sprites;

Interface

Type SpriteTyp=Record      {estructura de un bloque de datos de Sprite}
  Adr:Pointer;             {puntero a datos gráficos}
  dtx,dty:Word;            {altura y anchura en píxeles}
  px,py,                   {posición actual, opcional *}
  sx,sy:Integer;           {velocidad actual, opcional *}
End;
{*: opcional significa que las rutinas de sprites GetSprite y
PutSprite no hacen uso de estos datos, las variables sólo sirven para
simplificar el control por parte del programa principal}

Procedure GetSprite(Ofs,dtx,dty:Word;var zSprite:SpriteTyp);
{lee un sprite de vscreen-Offset ofs, con ancho dtx y alto dty,
 zsprite es el record de sprites, en el que se quiere guardar uno}

Procedure PutSprite(pg_ofs,x,y:Integer;qsprite:spritetyp);
{copia sprite de la memoria principal (long. y tamaño se toman de
qsprite) a la memoria de pantalla pág. pg en posición (x/y)}
```

```
Implementation
Uses ModeXLib;

Var i:Word;

Procedure GetSprite;
Var ppp:Array[0..3] of Byte;    {tabla con nº de píxeles a copiar}
                                {por Plane}
    Skip:word;                  {nº de bytes a saltar}
    Plane_Count:Word;           {contador de los planes ya copiados}
Begin
  GetMem(zsprite.adr,dtx*dty);  {alojar memoria principal}
  zsprite.dtx:=dtx;             {guardar ancho y alto en el sprite-record}
  zsprite.dty:=dty;

  i:=dtx shr 2;                 {nº de bloques de cuatro}
  ppp[0]:=i;ppp[1]:=i;          {corresponde nº mínimo de bytes a copiar}
  ppp[2]:=i;ppp[3]:=i;
  For i:=1 to dtx and 3 do      {guardar píxeles "sobrantes" en ppp}
    Inc(ppp[(i-1) and 3]);
  Plane_Count:=4;               {copiar 4 planes}
asm
  push ds
  mov di,word ptr zsprite {cargar puntero a bloque de datos}
  les di,[di]                   {cargar puntero a datos gráficos en es:di}
  lea bx,ppp                    {bx apunta a array-ppp}

  lds si,vscreen                {cargar puntero a imagen}
  add Ofs,si                    {añadir offset a los datos del sprite}
@lcopy_plane:                   {se recorre una vez por plane}
  mov si,ofs                    {cargar si con la direcc. incial de los da-
tos}
  mov dx,dty                    {cargar contador y con nº de líneas}

  xor ah,ah                     {borrar ah}
  mov al,ss:[bx]                {cargar al con la entrada ppp actual}
  shl ax,2                      {se mueven bloques de 4}
  sub ax,320                    {formar diferencia con 320}
  neg ax                        {hacer 320-ax de ax-320}
  mov skip,ax                   {guardar valor en skip}

@lcopy_y:                       {se ejecuta una vez por línea}
  mov cl,ss:[bx]                {cargar anchura del array ppp}
@lcopy_x:                       {se recorre una vez por punto}
  movsb                         {copiar byte}
  add si,3                      {al siguiente punto de este plane}
  dec cl                        {copiar todos los puntos de esta línea}
  jne @lcopy_x
```

```
  add si,skip              {al inicio de la siguiente línea}
  dec dx                   {copiar todas las líneas}
  jne @lcopy_y

  inc bx                   {posicionar a la sig. entrada ppp}
  inc ofs                  {posicionar a nuevo inicio de plane}
  dec plane_count          {copiar todos los planes}
  jne @lcopy_plane

  pop ds
End;
End;

Procedure PutSprite;
var plane_count,           {contador de los planes copiados}
    planemask:Byte;        {enmascara Write-plane en registro TS 2}
    Skip,                  {nº de bytes a saltar}
    ofs,                   {offset actual en memoria de pantalla}
    plane,                 {nº de plane actual}
    Anchura,               {anchura de bytes a copia en una línea,}
    dty:Word;              {altura}
    Fuente:Pointer;        {puntero a datos gráficos, si ds modif.}
    clip_lt, clip_rt:integer;  {nº de PIXEL sobrantes dcha. e izq.}
    clipact_lt,            {nº activo en plane actual}
    clipact_rt,            {BYTES sobrantes}
    clip_dn,clip_up:Word;  {nº de líneas sobrantes arriba y abajo}

    ppp:Array[0..3] of Byte;   {nº pixeles por plane}
    cpp:Array[0..3] of Byte;   {BYTES sobrantes por plane}

Begin
  if (x > 319) or          {representación inecesaria,}
  (x+qsprite.dtx < 0) or   {¿ya que no en imagen?}
  (y > 199) or
  (y+qsprite.dty < 0) then exit;
  clip_rt:=0;              {en caso normal sin clipping}
  clip_lt:=0;              {-> todas las variables clipping a 0}
  clip_dn:=0;
  clip_up:=0;
  clipact_rt:=0;
  clipact_lt:=0;
   with qsprite do begin
   if y+dty > 200 then begin   {1ª caso de clipping: abajo}
     clip_dn:=(y+dty-200);     {guardar líneas sobrantes}
     dty:=200-y;               {y reducir altura de sprite}
   End;
   if y<0 then begin           {2º caso de clipping: arriba}
     clip_up:=-y;              {guardar líneas sobrantes}
     dty:=dty+y;               {y reducir altura de sprite}
```

```
    y:=0;                    {Start-y es 0, porque borde superior}
  End;
  if x+dtx > 320 then begin   {3ª caso clipping: dcha.}
    clip_rt:=x+dtx-320;  {guardar píxeles sobrantes}
    dtx:=320-x;          {reducir anchura}
  End;
  if x<0 then begin        {4ª caso de clipping: izq.}
    clip_lt:=-x;         {guardar píxeles sobrantes}
    plane:=4-(clip_lt mod 4); {calcular nuevo startplane para col 0}
    plane:=plane and 3;  {reducir a 0..3}
    ofs:=pg_ofs+80*y+((x+1) div 4) - 1; {Ofs a bloque de 4 correcto}
    x:=0;                {comenzar representación en columna}
  End Else Begin         {dcha. no hay clipping?}
    plane:=x mod 4;      {cáclulo convencional del plane}
    ofs:=pg_ofs+80*y+(x div 4); {y Offset}
  End;
End;
Fuente:=qsprite.adr;     {puntero datos gráficos}
dty:=qsprite.dty;        {y guardar altura en var. local}
Anchura:=0;              {inicializar Anchura y Skip}
Skip:=0;

i:=qsprite.dtx shr 2;    {nº de bloques de 4 llanos}
ppp[0]:=i;ppp[1]:=i;     {corresponde al nº mínimo de bytes a copiar}
ppp[2]:=i;ppp[3]:=i;
For i:=1 to qsprite.dtx and 3 do{píxeles «sobrantes» en ppp}
  Inc(ppp[(plane+i - 1) and 3]);{añadir píxel comienzo startplane}

i:=(clip_lt+clip_rt) shr 2;
cpp[0]:=i;cpp[1]:=i;     {clipping por defecto: todas pág. 0}
cpp[2]:=i;cpp[3]:=i;
For i:=1 to clip_rt and 3 do  {si clipping dcha. guardar número}
  Inc(cpp[i-1]);         {correspondiente en Planes}
For i:=1 to clip_lt and 3 do  {si clipping dcha. guardar número}
  Inc(cpp[4-i]);         {correspondiente en Planes}

asm
  mov dx,3ceh            {GDC Register 5 (GDC Mode)}
  mov ax,4005h           {a Write Mode 0}
  out dx,ax
  push ds                {guardar ds}
  mov ax,0a000h          {cargar segmento destino VGA}
  mov es,ax

  lds si,Fuente          {fuente (puntero a datos gráficos) a ds:si}
  mov cx,plane           {crear máscara de planes inicial}
  mov ax,1               {desplazar bit 0 a la izq. en plane}
  shl ax,cl
```

```
  mov planemask,al              {guardar máscara}
  shl al,4                      {guardar en nibble superior}
  or planemask,al
  mov plane_count,4             {4 planes a copiar}
@lplane:                        {se recorre una vez por plane}
  mov cl,byte ptr plane         {cargar plane actual}
  mov di,cx                     {en di}
  mov cl,byte ptr ppp[di]       {cargar cx con nº correspondiente de ppp}
  mov byte ptr Anchura,cl       {calcular Skip de nuevo}
  mov ax,80                     {formar diferencia 80-Anchura}
  sub al,cl
  mov byte ptr skip,al          {escribir en Skip}

  mov al,byte ptr cpp[di]       {cargar ancho clipping específico del plane}
  cmp clip_lt,0                 {si no hay clipping izq., seguir con dcha.}
  je @rechts
  mov clipact_lt,ax             {guardar en clip_act_lt}
  sub Anchura,ax                {reducir anchura de bytes a copiar}
  jmp @clip_rdy                 {dcha. no hay clipping}
@rechts:                        {si izq. no hay clipping}
  mov clipact_rt,ax             {clipping todos los planes, en clip_act}
@clip_rdy:
  mov ax,Anchura                {calcular ancho total en bytes}
  add ax,clipact_rt
  add ax,clipact_lt
  mul clip_up                   {mul. con nº líneas del clipping superior}
  add si,ax                     {estos bytes no se representan}

  mov cx,Anchura                {cargar cx con anchura}
  or cl,cl                      {anchura 0, plane acabado}
  je @plane_listo

  mov di,ofs                    {offset destino en mem. pantalla a di}
  mov ah,planemask              {reducir máscara planes a bits [0..3]}
  and ah,0fh
  mov al,02h                    {y activar con reg. TS 2 (Write Plane Mask)}
  mov dx,3c4h
  out dx,ax
  mov bx,dty                    {inicializar contador y}
@lcopy_y:                       {bucle y, recorrer una vez por línea}
  add si,clipact_lt             {aumentar puntero fuente en clipping izq.}
  add di,clipact_lt             {también ptro. destino}
@lcopy:                         {bucle x recorrer una vez por punto}
  lodsb                         {obtener byte}
  or al,al                      {si 0, saltar}
  je @Valor0
  stosb                         {sino: activar}
@entry:
  loop @lcopy                   {y seguir bucle}
```

```
  add si,clipact_rt        {después de línea completa: clipping dcho.}

  dec bx                   {seguir contador y}
  je @plane_listo          {contador y = 0, siguiente plane}
  add di,skip              {sino saltar a inicio de línea}
  mov cx,Anchura           {reinicializar contador x}
  jmp @lcopy_y             {saltar de nuevo a bucle y}
@wert0:                    {color sprite 0:}
  inc di                   {saltar byte destino}
  jmp @entry               {y volver a bucle}
@plane_listo:              {aquí termina bucle y}
  mov ax,Anchura           {calcular anchura total en bytes}
  add ax,clipact_rt
  add ax,clipact_lt
  mul clip_dn              {mul. con nº líneas del clipping inferiror}
  add si,ax                {estos bytes no se representan}
  rol planemask,1          {enmascarar siguiente plane}
  mov cl,planemask         {plane 0 seleccionado?}
  and cx,1                 {(Bit 1 activo), entonces}
  add ofs,cx               {aumentar offset destino en 1 (bit cx 1)}
  inc plane                {nº plane (Index en ppp) seguir}
  and plane,3              {reducir a 0...3}
  dec plane_count          {¿4 planes copiados?, final}
  jne @lplane
  pop ds                   {restaurar ds, y adiós}
End;{asm}
End;

Begin
End.
```

Además de los dos procedimientos, la unidad también contiene las definiciones de tipo de un registro sprite, el cual recibe los datos más importantes de un sprite, creando para cada sprite individual una variable de este tipo. Lo primero que contiene es un puntero a los datos gráficos en sí en la memoria principal *(adr)*, siguiendo a continuación la anchura *(dtx)* y altura *(dty)*. A estas variables le suministra datos *Getsprite*, los cuales son utilizados por *Putsprite*, con lo cual se deberá evitar que otros procedimientos las borren.

Este bloque se puede ampliar libremente, según la aplicación que se le dé. Para una de las aplicaciones más comunes -el movimiento lineal- esta definición ya incluye cuatro variables que no son utilizadas por los procedimientos: *px* y *py* indican la posición actual del sprite y *sx* y *sy* su velocidad (anchura de paso durante cada generación de imagen) en dirección x e y. Pero, como ya hemos dicho, la utilización de estas variables depende del programa principal, lo único que hace la unidad es poner estas variables a disposición, con lo cual se pueden utilizar o ignorar.

Antes de poder representar los sprites en pantalla hay que crearlos. Aquí se vuelve a utilizar el formato GIF, dado que es el método más simple y compacto. Después de cargar una imagen que contiene los sprites, estos solamente tendrán que ser colocados en la memoria y ser guardados, para lo cual se utiliza el procedimiento *Getsprite* de esta unidad.

Este procedimiento parte de la base de que los sprites se encuentran en la imagen *vscreen* y que se conocen las coordenadas. Estas se traspasan como parámetros al llamar al procedimiento, pero no como coordenadas, sino como offset, lo cual no significa ningún problema, ya que este puede ser calculado según la fórmula ya conocida: *Offset := y * 320 + x* (*Vscreen* contiene la imagen en el formato del modo *13h*).

En este offset relativo a *vscreen*, *Getsprite* comienza con la lectura de los datos gráficos, definiéndose el tamaño mediante las siguientes variables: *dtx* indica la anchura y *dty* la altura del sprite. Como último parámetro el procedimiento *Getsprite* aún necesita una variable del tipo *spriteTyp*, en la cual guarda los datos del sprite.

Las variables son cargadas en el propio *getsprite* con sus valores correspondientes. Los valores de tamaño traspasados se guardan en el registro de sprite y se utilizan al mismo tiempo para el direccionamiento de la memoria.

El paso siguiente que hace el procedimiento es encargarse de las diferentes anchuras de los planos. Como ya hemos dicho antes, un sprite que no es divisible entre cuatro se reparte de forma diferente entre los diferentes planos. Por ello en este procedimiento lo primero que se hace es calcular el array con la anchura de los planos. De entrada, todos los planos tienen una anchura de *dtx/4*, repartiéndose el resto dentro de un bucle, partiendo del plano 0.

En la siguiente parte en ensamblador se encuentra la rutina de copiado en sí, que se compone de tres bucles anidados. Primero se cargan los punteros destino (*zsprite.adr*) y origen (*Vscreen+Ofs*) en *ES:DI y DS:DI*, señalando *BX* al array *ppp* en el segmento de stack (variable local).

El bucle exterior (*Icopy_plane*) se pasa una vez por cada plano, o sea cuatro veces, dando al registro *SI* cada vez un nuevo valor de inicio, a *DX* la altura del sprite y a la variable *Skip* el valor de separación de líneas válido para este plano.

Por cada línea de sprite se pasa una vez por el bucle (*icopy_Y*), el cual da a *CX* la anchura de línea actual antes del proceso de copia en sí. Después de copiar una línea, se suma el valor de salto de línea *skip* a *SI*, a fin de llegar al principio de la siguiente línea del sprite; este bucle es ejecutado *DX* veces, ya que *DX* obtuvo en el bucle del plano la altura del sprite.

El bucle interior no hace nada más que copiar un byte. Pero dado que los datos de un plano están en el formato del modo *13h*, el cual utiliza *vscreen*, siempre tienen una separación de 4 píxeles, *SI* aún deberá incrementarse en 3, ya que *movsb* solamente incrementa en 1.

El segundo procedimiento (*Putsprite*) de esta unidad es bastante similar en su estructura al procedimiento *Getsprite*, sólo que un poco más complicado. A este procedimiento se le pasan cuatro parámetros: primero el offset de la página de pantalla actual, normalmente *vpage*, a continuación las coordenadas X e Y del sprite, haciendo referencia éstas a la esquina superior izquierda del mismo. Finalmente sigue el *sprite record* que ya conocemos de *Getsprite*, del cual *Putsprite* extrae las informaciones necesarias para la representación.

En las coordenadas ya se aprecia una característica especial de *Putsprite* frente a *Getsprite:* ambas variables están definidas como integer, por lo que también aceptan valores negativos. En el caso de valores negativos la esquina superior izquierda del sprite estará situada fuera de la pantalla, por lo cual habrá que efectuar un clipping o bien arriba o bien a la izquierda; lo mismo sucede con coordenadas grandes que hacen desaparecer el sprite por el borde derecho y/o inferior.

Por ello se trata primero el clipping dentro del procedimiento. Los casos más simples en los que hay efectuar un tratamiento especial, pero que estrictamente no son un clipping, son aquellos en los cuales el sprite no es visible en la pantalla. En estos casos (primera orden *If*) se interrumpe simplemente la representación mediante una orden *exit* y sigue el programa de forma normal.

Si el sprite es visible parcialmente, el procedimiento pasa a su segunda parte, en la cual se calcula el clipping. Primero se sitúan todas las variables que tienen que ver con el clipping a 0, lo cual significa una representación total, y a continuación se prepara el clipping para cada borde de la pantalla.

Primero se trata el caso más simple, o sea el clipping en el borde inferior (*if y+dty > 200*). En este caso, en principio, solamente hay que reducir la altura a la zona visible realmente. Cada sprite se trata cuatro veces de forma que, en este caso de clipping, hay que encargarse de que, a pesar de ello, después de cada plano el puntero marque el principio de los siguientes datos de plano. Para ello se utiliza la variable *clip_dn*, que contiene el número de líneas que deben ser saltadas al final de cada plano, ya que no son representadas.

El clipping en el borde superior también se puede tratar de una forma simple: en este caso también se apunta el número de líneas que se deben saltar (esta vez en *clip_up*) y se reduce la altura. Además hay que colocar *y* en la nueva línea de inicio, o sea, a *0*.

La cosa se complica un poco con el borde derecho: en el clipping horizontal hay que tener en cuenta siempre el modelo de planos, que se encarga, de forma similar a las anchuras diferentes de los planos, de que el clipping sea diferente para cada plano. Este hecho se tiene en cuenta en el siguiente bucle. Aquí simplemente se guarda el número de píxeles (en *clip_rt)*, de la misma forma que en el clipping inferior y superior y se reduce la anchura.

En el borde izquierdo se actúa de forma diferente: también se guarda el número de bytes, pero se mantiene la anchura y se iguala dentro de la rutina de copiado, a fin de garantizar la sincronización con el clipping vertical. En este caso hay que volver a calcular el plano de inicio. Nos podemos imaginar, por ejemplo, un sprite que sobresalga tres píxeles por el borde izquierdo.

En la representación se sigue comenzando con el bloque de datos del primer plano dentro del sprite, en el cual se encuentra en este caso la columna -3. Pero dado que la columna -3 no se muestra, se utiliza la primera columna visible del plano, o sea, en este caso la columna *-3+4=1*. Esto se tiene en cuenta en el cálculo del plano. El offset también se tiene que volver a calcular, ya que en el caso de un valor X negativo marcaría a un punto de la línea anterior. Como coordenada X se asigna el valor 0, de la misma forma que en el borde superior.

Los valores de plano y de offset modificados solamente son válidos para el clipping en el borde izquierdo, pudiéndose calcular en todos los demás casos los valores de la forma normal, como se ha hecho hasta ahora en el modo X. Después de inicializar algunas variables, se calcula de la misma forma que en *getsprite* el array *ppp* de las diferentes anchuras de los planos. Ahora falta determinar de una manera bastante similar la tabla *cpp*, la cual contiene los valores del clipping para el borde derecho e izquierdo. Un clipping en ambos lados no es posible, pero tampoco tendría mucho sentido, por lo que incluirlo solamente complicaría más la cosa.

Para todos los planos la base es la anchura del clipping (*clip_lt+clip_rt)* dividida entre 4, a fin de determinar la cantidad de bloques de cuatro. A este valor se le añade el específico de cada plano, que se suma en una construcción muy similar al bucle *ppp*. Aquí lo importante es la diferencia entre el borde derecho e izquierdo. En el borde derecho el resto se reparte partiendo del plano 0 (*cpp[i-1])*, mientras que en el borde izquierdo comienza en el plano 3 (*cpp[4-i])*.

La parte en ensamblador establece primero el modo de escritura 0 necesario, que permite un direccionamiento directo de los puntos individuales. Después de cargar el segmento destino y el puntero origen, deberá determinarse la máscara para la selección del plano actual. La máscara, que deberá ser enviada a *TS*, deberá colocar un bit correspondiente para los planos a tratar, en este caso sólo uno, por lo que el valor numérico entre 0 y 3 deberá ser convertido en el bit correspondiente.

0 se convierte en *0001b*, 1 en *0010b*, 2 en *0100b* y 3 en *1000b*. Esta máscara se guarda en *planemask*, para rotarla después de la copia de cada plano en una posición hacia la izquierda.

La variable *plane_count* tiene una tarea completamente diferente: se trata de un simple contador que se encarga que el bucle de plano sea ejecutado cuatro veces. Este bucle *@lplane* comienza con la carga de la anchura actual del array *ppp* hacia *anchura* y el cálculo de la anchura de salto en la variable *Skip*.

Mediante la tabla *cpp* se prepara ahora el clipping. Si éste se produce en el borde derecho, se carga simplemente la variable *clipact_rt* con el valor *cpp* y estará a disposición después de cada paso por el bucle, a fin de saltarse el número de bytes correspondiente. En el borde izquierdo se utiliza de forma similar la variable *clipact_lt*, debiéndose reducir al mismo tiempo la anchura, lo cual no puede suceder en la parte en Pascal debido al plano de inicio modificado, y que tiene que efectuarse aquí para cada plano.

Después de estas preparaciones se trata el primer caso de clipping, multiplicando la anchura total de una línea incluidos el clipping derecho e izquierdo por la cantidad de líneas que hay que clippear en el borde superior. Estos bytes simplemente se saltan mediante una adición al registro *SI*.

En este punto se incluye una comprobación de la anchura del plano, la cual interrumpe el plano cuando la anchura es de 0, ya que no debe ser copiado. Esta comprobación no solamente es necesaria en las anchuras totales de sprite *0*, lo cual sería bastante estúpido. Debido al modo X los planos tienen anchuras diferentes, por lo que los sprites con una anchura total de menos de 4 puntos por lo menos faltarán en un plano.

Después de colocar el plano actual mediante el registro 2 del secuenciador del timing puede comenzar el siguiente bucle interior *@lcopy_y*, que genera primero para cada línea el clipping izquierdo incrementando el puntero origen y destino, copiando luego la línea y tratando finalmente también el clipping derecho (*add SI, clipakt_rt*). A parte se mueve el puntero destino al principio de la siguiente línea y se carga el contador *XCX* con la anchura. El bucle de copia en sí (*@locpy*) simplemente mueve un byte desde la dirección origen a la dirección destino, siempre y cuando no sea un 0, lo cual representaría un pixel de fondo y simplemente se salta.

Después de haber liquidado un plano (etiqueta *@plane_listo*) se calcula de la misma forma que en el clipping superior el inferior y se rota la máscara de plano. En esta rotación, en la mayoría de los casos, se produce un desbordamiento por cada sprite, lo que significa que después del plano 2 se copia el 0. En este caso también hay que incrementar el offset, a fin de que no se copie una columna a la izquierda

del sprite. Esto se puede capturar mediante una combinación *CMP-JNE* (que se debería evitar por su lentitud), pero aquí se utiliza un procedimiento más rápido: en el caso de un desbordamiento el enmascaramiento de plano (*planemask)* salta de *1000b a 0001b*, simplemente se desenmascara el bit 0 y se va sumando cada vez a un offset, de forma que ello solamente produce un resultado cuando la máscara es *0001b*, evitándose todos los demás casos. El bucle de plano puede finalizarse después de actualizar los dos contadores de plano *Plane y Plane_count*.

Un último comentario sobre el código Inline de estas rutinas: siempre es mejor sacar las partes en ensamblador a módulos *ASM* y ensamblarlos de forma separada, para poder aprovechar así las grandes ventajas de un ensamblador "de primer orden" (macros, estructuras, órdenes REP, etc.) Pero las partes en ensamblador precisan muchas variables de las partes en Pascal, de forma que aquí se ha utilizado la orden interna *ASM* del Pascal. La utilización de estas rutinas queda explicada en el programa *SPRT_TST.PAS*:

```
Uses Crt,Gif,ModeXLib,Sprites;

Const Numero_Sprites=3;  {nº de sprites empleados en el programa}

Var Sprite:Array[1..Numero_Sprites] of SpriteTyp;
                         {records de datos de los sprites}
  i:Word;                {contador}

Begin
  Init_ModeX;            {activar modo X}
  LoadGif('sprites');    {cargar imagen con 3 sprites}
  GetSprite(62 +114*320,58,48,Sprite[1]);  {coordenadas (62/114), an-
cho 58*48}
  GetSprite(133+114*320,58,48,Sprite[2]); {(133/114), 58*48}
  GetSprite(203+114*320,58,48,Sprite[3]); {(203/114), 58*48}
                         {cargar los 3 sprites}
  LoadGif('phint');      {cargar imagen de fondo}
  p13_2_ModeX(48000,16000);    {y copiar a la página de fondo}
  With Sprite[1] do Begin {coordenadas y velocidades}
    px:=160;py:=100;     {de los 3 sprites (aleatorio)}
    sx:=1;sy:=2;
  End;
  With Sprite[2] do Begin
    px:=0;py:=0;
    sx:=1;sy:=-1;
  End;
  With Sprite[3] do Begin
    px:=250;py:=150;
    sx:=-2;sy:=-1;
  End;
```

```
  Repeat
    CopyScreen(vpage,48000);    {imagen de fondo a página actual}
    For i:=1 to Numero_Sprites do{recorrer para los 3 sprites}
      With Sprite[i] do Begin
        Inc(px,sx); Inc(py,sy); {movimiento}
        If (px < -dtx div 2)    {¿borde dcho. o izq.? -> invertir}
          or (px > 320-dtx div 2) Then sx:=-sx;
        If (py < -dty div 2)    {¿borde sup. o inf.? -> invertir}
          or (py > 200-dty div 2) Then sy:=-sy;
        PutSprite(vpage,px,py,Sprite[i]);   {dibujar sprite}
      End;
    switch;                 {conmutar a página calculada}
    WaitRetrace;            {pantalla sólo se puede modificar después}
  Until KeyPressed;         {del siguiente retrazado}
  ReadLn;
  TextMode(3);
End.
```

Después de cargar los sprites, éstos se colocan en posiciones (aleatorias) y a diferentes velocidades. El bucle de repetición representa el principio fundamental de la representación de sprites: borrar el fondo (*CopyScreen*), mover los sprites, mostrar los sprites. La llamada a *WaitRetrace* está situada, en este caso, detrás de la instrucción *Switch*, ya que el cambio a una nueva página se efectúa con un retrace de retardo. Por ello aquí se aplica una lógica diferente a la utilizada hasta ahora: primero se envía la orden de cambio a la VGA. Si ahora se espera al retrace, se estará seguro que en este momento se efectúa el cambio y que se puede preparar la siguiente página. Por desgracia las rutinas de sprites precisan grandes cálculos, por lo que en ordenadores poco potentes este programa de demostración con tres sprites ya puede ser demasiado complicado. En ese caso lo que hay que hacer es reducir su número.

7.5 Para movimientos realistas - escalado

Cuando un personaje de una aventura gráfica camina hacia atrás debe ir perdiendo tamaño debido a los puntos de fuga, a fin de crear una imagen tridimensional. Con los medios que conocemos hasta ahora, solamente existe una posibilidad: para cada etapa intermedia se define un sprite que ha sido convertido al tamaño adecuado con un programa de dibujo. Este es el método más rápido, pero al mismo tiempo el que ocupa más memoria. Por ello en muchas ocasiones será conveniente crear el escalado de forma automática en el programa y trabajar solamente con el sprite original.

Para solucionar este problema existen algoritmos matemáticos generales que proyectan cada punto de forma individual a su nueva posición. No creo que haga falta explicar que esto es muy lento. Pero existe un procedimiento mucho más simple, cuyo resultado es idéntico al matemático, pero que se basa en una teoría diferente.

Como ejemplo vamos a explicar el escalado en dirección Y, que en nuestro ejemplo genera un logotipo que gira. Un escalado en dirección X funciona, en principio, de la misma forma, de manera que no hace falta explicarlo.

Si se desea escalar un sprite por un factor entero, por ejemplo 1/2, simplemente se representa una línea de cada dos, es decir, se salta cada segunda línea. En una reducción a un tercio se utilizaría de forma análoga solamente una de cada tres líneas. Este procedimiento se puede aplicar a cualquier fracción, calculándose a continuación, y también en el programa, en porcentajes.

De lo que se trata es de ir comprobando en cada momento después de cuántas líneas hay que saltarse una. Para ello se le añade a la coordenada Y una parte decimal, que en la representación se redondea a una coordenada entera. Este procedimiento parece lógico si se mira la pantalla desde un punto de vista matemático, en el cual no solamente existen líneas enteras, sino cualquier valor intermedio que se utiliza durante el escalado. Pero dado que la pantalla tiene un número limitado de líneas, estas líneas intermedias tendrán que ser representadas sobre las líneas existentes realmente.

Hasta aquí la teoría, pero ¿cómo funciona todo esto en la práctica? Aquí entra en acción el cálculo con coma fija (suma y resta), ya que es la única forma rápida. La coordenada Y relativa dentro del sprite se divide en una parte anterior a la coma, que no se guarda explícitamente en una variable, y una parte decimal (en el programa en la variable *rel_y)*. Ahora después de cada línea se suma una constante (*add_y* en el programa) a la parte decimal. Cuando esta supera el valor 100, que corresponde a un desbordamiento de la parte anterior a la coma, se extrae la siguiente consecuencia: se incrementa la parte anterior a la coma, lo cual significa que se salta la siguiente línea, reduciéndose a parte la parte decimal en 100.

La constante sigue la siguiente norma: cuanto mayor sea el factor de escalado (más cerca del original) menor será (ya que se elimina una línea en menos ocasiones), calculándose simplemente como factor de escalado 100. Un ejemplo: un sprite se escala al 60%, o sea que la constante de suma es de 40. Después de la primera línea la parte decimal será de 40, después de la segunda de 80. En ambos casos no sucede nada más, pero después de la tercera línea la parte decimal pasará con 120 del límite de 100. Se restablece a 20 y se salta la siguiente línea, repitiéndose lo mismo después de otras dos líneas con una parte decimal de 100, comenzando de nuevo el juego en 0.

Los saltos en sí se pueden llevar a cabo de dos formas, por un lado se puede saltar la línea en los datos origen, y por otro simplemente se puede mantener el puntero destino y escribir la siguiente línea encima de la anterior. Nosotros preferimos este segundo método: en éste se representa cada línea, aunque no todas sean visibles. A causa de ello la velocidad de representación es siempre la misma, independien-

temente del grado de escalado. Este procedimiento permite, además, la generación de sprites reflejados, tal como nos muestra el programa *SCAL_TST.PAS*:

```
Uses Crt,Sprites,ModeXLib,Gif,Tools;

Procedure PutScalSprt(pg_ofs,x,y,scale_y:Integer;qsprite:spritetyp);
var planecount,              {contador de los planes ya copiados}
    planemask:Byte;          {enmascara Write-Plane en registro TS 2}
    Skip,                    {nº de bytes a saltar}
    ofs,                     {offset actual en mem. pantalla}
    plane,                   {nº de plane actual}
    Anchura,                 {anchura de bytes a copiar en una línea}
    dty:Word;                {altura}
    Fuente:Pointer;          {puntero a datos gráficos, si ds modificado}

    ppp:Array[0..3] of Byte;     {nº pixeles por plane}
    rel_y,                   {parte decimal de la pos. rel y}
    add_y:Word;              {parte decimal del sumando}
    direction:Integer;       {dirección de movimiento (+/- 80)}
    i:Word;                  {contador local de bucle}
Begin
  if (x + qsprite.dtx > 319)    {Clipping ? interrumpir}
  or (x < 0)
  or (y + qsprite.dty*scale_y div 100 > 199) or (y < 0) then exit;
  add_y:=100-abs(scale_y);      {calcular sumandos}
  if scale_y < 0 then direction:=-80 else direction:=80;
                           {determinar dirección}
  Fuente:=qsprite.adr;     {puntero datos gráficos}
  dty:=qsprite.dty;        {cargar variable local de altura}
  plane:=x mod 4;          {Start-Plane}
  ofs:=pg_ofs+80*y+(x div 4);  {y calcular offset}
  Anchura:=0;              {inicializar Anchura y Skip}
  Skip:=0;

  i:=qsprite.dtx shr 2;    {nº de bloques de 4 llanos}
  ppp[0]:=i;ppp[1]:=i;     {corresponde a número mínimo de bytes a co-
piar}
  ppp[2]:=i;ppp[3]:=i;
  For i:=1 to qsprite.dtx and 3 do{píxeles «sobrantes» en ppp}
    Inc(ppp[(plane+i - 1) and 3]);{añadir píxel desde startplane}
asm
  push ds                  {guardar ds}
  mov ax,0a000h            {cargar segmento destino VGA}
  mov es,ax

  lds si,Fuente            {fuente (ptro. a datos gráficos) a ds:si}
  mov cx,plane             {crear máscara de planes inicial}
  mov ax,1                 {desplazar bit 0 a la izq. en plane}
```

```
  shl ax,cl
  mov planemask,al          {guardar máscara}
  shl al,4                  {guardar también en nibble superior}
  or planemask,al
  mov planecount,4          {copiar 4 planes }
@lplane:                    {se recorre una vez por plane }
  mov cl,byte ptr plane     {cargar plane actual}
  mov di,cx                 {en di}
  mov cl,byte ptr ppp[di]   {cargar cx con nº ppp actual}
  mov byte ptr Anchura,cl   {calcular Skip de nuevo}
  mov ax,direction          {formar diferencia Direction-Anchura}
  sub ax,cx
  mov skip,ax               {y escribir en skip}

  mov rel_y,0               {inicio de nuevo en y=0,0}

  mov cx,Anchura            {cargar cx con anchura}
  or cl,cl                  {anchura 0, plane listo}
  je @plane_listo

  mov di,ofs                {offset destino en mem. pantalla a di}
  mov ah,planemask          {reducir máscara de planes a 0...3}
  and ah,0fh
  mov al,02h                {fijar con registro TS 2 (Write Plane Mask)}
  mov dx,3c4h
  out dx,ax
  mov bx,dty                {inicializar contador y}
@lcopy_y:                   {bucle y, recorrer una vez por línea}
@lcopy_x:                   {bucle x, recorrer una vez por punto}
  lodsb                     {obtener byte}
  or al,al                  {si 0, saltar}
  je @Valor0
  stosb                     {si no: activar}
@entry:
  loop @lcopy_x             {y seguir bucle}

  mov ax,rel_y              {sumando a parte decimal}
  add ax,add_y
  cmp ax,100                {aumentada mantisa?}
  jb @noaddovfl             {no, seguir}
  sub ax,100                {sino reponer parte decimal}
  sub di,direction          {y a la siguiente/anterior línea}
@noaddovfl:
  mov rel_y,ax              {y escribir a la parte decimal}

  dec bx                    {seguir contador y}
  je @plane_listo           {contador y = 0, siguiente plane}
  add di,skip               {sino saltar a la siguiente línea}
  mov cx,Anchura            {inicializar contador x}
```

```
  jmp @lcopy_y              {saltar de nuevo a bucle y}
@Valor0:                    {color sprite 0:}
  inc di                    {saltar byte destino}
  jmp @entry                {y volver al bucle}
@plane_listo:               {bucle y terminado}

  rol planemask,1           {enmascar siguiente plane}
  mov cl,planemask          {plane 0 seleccionado?}
  and cx,1                  {(Bit 1 activo), entonces}
  add ofs,cx                {aumentar offset destino en 1 (Bit 1 !)}
  inc plane                 {incr. nº plane (Index en ppp)}
  and plane,3               {reducir a 0...3}
  dec planecount            {4 planes copiados?, fin}
  jne @lplane
  pop ds                    {restaurar ds, y adiós}
End;{asm}
End;

Var Logo:SpriteTyp;
    Seno:Array[0..99] of Word;
    Altura:Integer;
    i:Word;

Begin
  Init_ModeX;               {activar modo X}
  LoadGif('sprites');       {cargar imagen con logo}
  GetSprite(88+ 6*320,150,82,Logo);  {inicializar logo}
  LoadGif('phint');         {cargar imagen de fondo}
  p13_2_ModeX(48000,16000);  {y copiar a pág. de fondo}
  Sin_Gen(Seno,100,100,0);  {calcular senos}
  I:=0;                     {índice en seno a 0}
  repeat
    Inc(i);                 {incr. índice}
    Altura:=Integer(Seno[i mod 100]); {obtener altura del seno}
    CopyScreen(vpage,48000);    {borrar fondo}
    PutScalSprt(vpage,85,100-Altura *84 div 200,Altura,Logo);
                            {copiar sprite escalado a pág. actual}
    Switch;                 {pasar a esta página}
    WaitRetrace;            {esperar retrazado}
  Until KeyPressed;
  ReadLn;
  TextMode(3);              {modo normal de texto}
End.
```

En este programa se genera un logotipo que gira sobre un fondo, modificándose la altura de forma periódica mediante una tabla senoidal (la coordenada Y de un movimiento circular aparece desde el lado como una onda senoidal). El marco del

programa se corresponde con la anterior demo de sprites, siendo igual la administración del sprite, lo que quiere decir que cualquier sprite puede ser representado con *Putsprite* y escalado con *PutScalSprt* y ser representado en la pantalla.

El procedimiento *PutScalSprt* tiene un carácter de demostración, por lo cual no ha sido ubicado en su propia unidad. Además, hemos eliminado el clipping por razones de velocidad, algo que tampoco es imprescindible en logotipos en rotación o similares. Además de los parámetros normales, se le comunica al procedimiento mediante *scale_y* el porcentaje del tamaño original que tendrá el sprite escalado. Un valor negativo significa un reflejo sobre el eje X, lo cual permite una rotación confortable.

La variable *Add_Y* se inicializa tal como hemos dicho con *100-Scale_Y*, utilizándose para ello el valor del factor de escalado. La reflexión se controla mediante la siguiente variable *direction*, que es de 80 en valores de escalado positivos y en otros casos de -80 y que indica casi la anchura total de la línea. También se tiene en cuenta en el cálculo de la variable *Skip*, donde la anchura de sprite no es de 80, sino que se resta de la variable *direction*. Cuando *direction* es negativa, *Skip* también se vuelve negativa, lo cual corresponde a un movimiento a la línea anterior.

La parte de escalado en sí se encuentra directamente después del bucle *@lcopy_x*, por lo que se ejecuta después de cada línea representada. Aquí se efectúa la suma de la parte decimal (posterior a la coma), donde se decide si se produce un desbordamiento. En ese caso se vuelve a reducir la parte decimal en 100 y se salta de vuelta a la línea ya tratada en la memoria de vídeo (según el valor de *direction* hacia atrás o hacia adelante).

8. La tercera dimensión - programación gráfica 3D

Todos los efectos mostrados hasta ahora tienen algo en común: sólo trabajan en dos dimensiones. Si se quieren crear imágenes más reales, no se puede evitar la tercera dimensión, la profundidad. El problema que se presenta es la representación en el monitor, que sigue siendo de dos dimensiones, y que sólo puede simular la información de profundidad mediante trucos como la perspectiva de puntos de fuga y el sombreado mediante luces.

Estos trucos necesitan una buena dosis de matemáticas, mejor dicho, de geometría. Transformaciones, proyecciones, degradados, etc., que se piden de forma masiva para una presentación impresionante, necesitan cálculos muy complejos. Estos se han de optimizar según aspectos matemáticos e informáticos, para que se obtenga una velocidad razonable. El lenguaje de estos procedimientos es por ello el ensamblador, ya que un lenguaje de alto nivel genera estructuras demasiado complejas y, por consiguiente, lentas.

A lo largo de este capítulo conocerá la aplicación práctica de los métodos teóricos con ayuda de programas cortos. Estos programas se basan todos en los mismos módulos en ensamblador (3DASM.ASM, POLY.ASM, BRES.ASM, TEXTURE.INC), que contienen tanto funciones independientes de representación (transformaciones, etc.) como las especiales (sombreado mediante luces) que se activan mediante variables globales. Este control a través de las variables globales no parece, en primera instancia, muy elegante, pero ahorra longitud de código y, sobre todo, tiempo de cálculo.

8.1 Matemáticas para fanáticos de los gráficos

Si ya conoce la geometría analítica, puede olvidar este capítulo. Pero quien no tenga estos conocimientos básicos, podrá conocer aquí conceptos como vector, determinante y producto matricial, que son importantes para la posterior comprensión.

El vector

Lo que es el número para el álgebra, lo es el vector para la geometría. Un vector se puede imaginar como una flecha (perdón a todos los matemáticos), que apunta en una dirección determinada y que tiene una longitud determi-

nada. Un vector es libremente desplazable en el espacio y está definido unívocamente por sus componentes, que indican su longitud y dirección en los ejes x, y y z.

En esta definición no se encuentra ninguna indicación sobre en qué punto exacto del espacio se encuentra el vector. Esto es diferente en los llamados vectores de origen. Estos siempre parten del origen del sistema de coordenadas (*0/0*) y por ello determinan un punto concreto.

En el método de representación de vectores las componentes del mismo se escriben una encima de otra.

$$\vec{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \quad \text{o} \quad \vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

$$\text{p.e.: } \vec{a} = \begin{pmatrix} 2 \\ 3 \\ -5 \end{pmatrix}$$

Figura 11: Representación matemática de un vector

Calcular con vectores

Como hemos dicho, un vector contiene tanto informaciones de dirección como de longitud. Precisamente esta última se puede obtener muy fácilmente de las componentes, aplicando el teorema tridimensional de Pitágoras.

$$a = |\vec{a}| = \left\| \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \right\| = a_1^2 + a_2^2 + a_3^2$$

$$\text{p.e.:} \left\| \begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix} \right\| = 3^2 + (-2)^2 + 1^2 = 14$$

Figura 12: Longitud de un vector

Los vectores se suman, sumando sus componentes. Geométricamente la suma significa una colocación secuencial de los dos vectores.

La resta funciona de forma similar. Las componentes se restan, es decir, geométricamente se suma la inversión del segundo vector al primero.

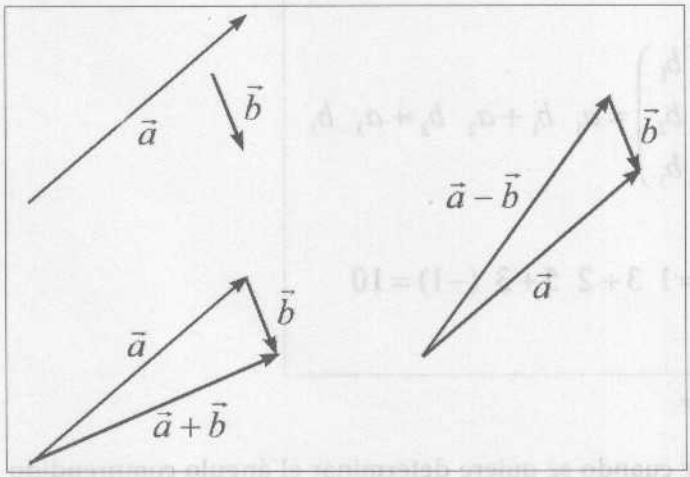

Figura 13: Resta de vectores

En la multiplicación existen varios métodos con resultados totalmente diferentes:

En el escalado (multiplicación S) se multiplica un vector con un número, con lo que sólo se modifica su longitud. Una multiplicación S con un número negativo invierte además la dirección.

$$(-3)\begin{pmatrix}1\\-2\\3\end{pmatrix}=\begin{pmatrix}-3\\6\\-9\end{pmatrix}$$

Figura 14: Multiplicación de vectores

La segunda posibilidad, la multiplicación escalar, multiplica dos vectores el uno con el otro, y el resultado es un número (un escalar). Existen dos definiciones:

1. $\vec{a}\ \vec{b} = a\ \ b\ \cos\alpha$

α: Ángulo entre dos vectores

2. $\vec{a}\ \vec{b} = \begin{pmatrix}a_1\\a_2\\a_3\end{pmatrix}\begin{pmatrix}b_1\\b_2\\b_3\end{pmatrix} = a_1\ \ b_1 + a_2\ \ b_2 + a_3\ \ b_3$

z.B.: $\begin{pmatrix}1\\2\\3\end{pmatrix}\begin{pmatrix}3\\5\\-1\end{pmatrix} = 1\ \ 3 + 2\ \ 5 + 3\ \ (-1) = 10$

Figura 15: Multiplicación escalar

Esto se puede emplear cuando se quiere determinar el ángulo comprendido entre dos vectores, sustituyendo la primera ecuación en la segunda, y despejando el *cos* α.

$$\cos\alpha = \frac{\vec{a}\ \vec{b}}{a\ b}$$

$$= \frac{a_1\ b_1 + a_2\ b_2 + a_3\ b_3}{a\ b}$$

Figura 16: Ángulo de un vector

Como último método de multiplicación existe el producto matricial o vectorial. Se multiplican los vectores el uno con el otro, y el resultado es a su vez otro vector. Este vector es perpendicular a los dos vectores multiplicados, por lo que el producto vectorial sólo tiene sentido en un espacio tridimensional. Se define de la siguiente forma:

$$\vec{a}\quad \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2\ b_3 - a_3\ b_2 \\ a_3\ b_1 - a_1\ b_3 \\ a_1\ b_2 - a_2\ b_1 \end{pmatrix}$$

$$\text{p.e.:} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \begin{pmatrix} 3 \\ 5 \\ -1 \end{pmatrix} = \begin{pmatrix} 2\ (-1) - 3\ 5 \\ 3\ 3 - 1\ (-1) \\ 1\ 5 - 2\ 3 \end{pmatrix} = \begin{pmatrix} -17 \\ 10 \\ -1 \end{pmatrix}$$

Figura 17: Producto vectorial

8.2 Representar cuerpos 3-D en 2-D

Mientras no existan dispositivos de salida 3D a precios razonables, sólo nos queda la representación plana para todos los cálculos tridimensionales. Para ésta existen varios métodos, que van desde la sencilla proyección paralela hasta complejos algoritmos de Raytracing. Pero este último método queda

reservado para los llamados "Renderers", que pueden tomarse unos minutos de cálculo para cada imagen.

El método más sencillo para convertir informaciones de profundidad que el monitor no puede representar, es ignorarlas. Las coordenadas bidimensionales de los vértices de cada superficie se obtienen de las coordenadas x e y de la definición tridimensional. En este método las rectas paralelas del espacio tridimensional aparecen como rectas paralelas en la pantalla. De ahí el nombre proyección paralela.

Esto tiene algunas ventajas (y se emplea por ejemplo en la arquitectura), pero tiene poco que ver con la realidad. Al fin y al cabo, los objetos alejados crean una imagen más pequeña en la retina del ojo. Esta propiedad se ha de simular mediante reducción, no ignorando la profundidad, sino empleándola para la disminución de la imagen. El algoritmo necesario se puede ver fácilmente observando las ecuaciones de los rayos ópticos:

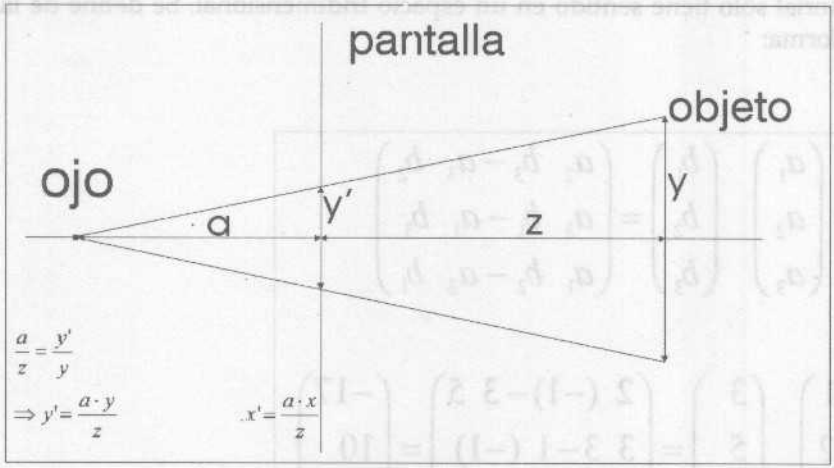

Figura 18: Las ecuaciones de rayos ópticos

Aquí se considera el mundo tridimensional como situado detrás de la pantalla. De este mundo no se ve nada en principio, pero la imagen es visible en la pantalla. Todos los rayos que parten del objeto, han de llegar finalmente al ojo, de modo que en la imagen éste es el origen. Cada coordenada del objeto (aquí por ejemplo para las coordenadas y de las esquinas superior e inferior) se ha de convertir en una coordenada de pantalla bidimensional (aquí y') Para ello se emplea la ecuación de rayos ópticos que crea una relación unívoca entre y, y', a (distancia del ojo y la pantalla) y z (profundidad del objeto).

Se puede comprender fácilmente que en este algoritmo una profundidad mayor (z) disminuye el ángulo entre los rayos y con ello la imagen en pantalla, de modo que así se obtiene la perspectiva de punto de fuga deseada. Este punto de fuga, en el que todas las líneas paralelas convergen a lo lejos, se encuentra siempre en el eje z en este método y no es libremente emplazable (lo que tampoco es necesario para la mayoría de aplicaciones).

En los cálculos mediante esta fórmula se puede ahorrar mucho tiempo de cálculo, si se emplea una potencia de dos (por ejemplo *128*) para la distancia *a* (aleatoria de todos modos). En ese caso se puede acelerar la multiplicación mediante desplazamientos (*SHL/SHR*).

8.3 Deformar objetos - transformaciones

En los menos de los casos uno se conformará con una imagen inalterable al programar mundos tridimensionales. Precisamente el movimiento es el que da ese toque real. Un dado rotando, con dibujos en sus caras hace la cosa mucho más interesante que una imagen estática que es mucho más sencilla de crear con un programa de dibujo. Los movimientos se componen fundamentalmente de dos cosas: traslación y rotación. El escalado puede añadirse en determinadas circunstancias.

La traslación no es otra cosa que un desplazamiento en una dirección determinada, por ejemplo el movimiento a lo largo de un pasillo. Matemáticamente la traslación se basa en una suma vectorial: el desplazamiento es determinado por un vector, el vector de traslación. Este vector simplemente se suma a todos los vectores originales (coordenadas de punto) del objeto a desplazar.

El movimiento del observador se obtiene de la misma forma, aunque el modo de observación sea al revés. Si el observador se quiere mover una unidad en dirección z, simplemente se desplaza todo el mundo 3D una unidad en la dirección negativa de la z.

La rotación es algo más complicada, sobre todo si se quiere calcular con matrices a toda costa, lo que frecuentemente se recomienda. Las matrices resumen las operaciones matemáticas necesarias en una especie de tabla: traslaciones, rotaciones, escalados, todo tiene su matriz. Esto tiene la ventaja de que se pueden resumir varias matrices y con ello se puede ahorrar algo de tiempo de cálculo - siempre que desde el principio esté claro qué transformaciones se han de realizar y en qué orden: pero como esto ocurre muy pocas veces, las matrices de rotación sólo serán mencionadas al margen.

En la rotación se ha de distinguir fundamentalmente alrededor de cuál de los tres ejes se ha de rotar ya que se realizan diferentes tipos de operaciones. Para la rotación alrededor del eje *x* se aplica el siguiente algoritmo:

```
x'= x
y'= y*cos(a)-z*sin(a)
z'= y*sin(a)+z*cos(a)
```

Y para quien le interese, esta es la matriz correspondiente:

```
1 0    0
0 cos(a)     -sin(a)
0 sin(a)     cos(a)
```

Alrededor del eje *y*:

```
x'=x*cos(a)+z*sin(a)
y'=y
z'=-x*sin(a)+z*cos(a)
```

Y la matriz correspondiente:

```
cos(a)  0  sin(a)
0  1  0
-sin(a) 0  cos(a)
```

Alrededor del eje z:

```
x'= x*cos(a)-y*sin(a)
y'= x*sin(a)+y*cos(a)
z'= z
```

Y la matriz correspondiente:

```
cos(a)   -sin(a)  0
sin(a)   cos(a)   0
0        0        1
```

Estas fórmulas de momento sólo consideran la rotación alrededor de los ejes de coordenadas. Mediante la combinación de varias rotaciones se puede formar cualquier recta que pase por el origen de coordenadas. Si también se quieren evitar las limitaciones de la recta del origen, se ha de unir la rotación a una traslación.

Para ello se desplaza la rotación en el mundo, de forma que el punto alrededor del cual se está girando se encuentre en el origen. Después, la rotación se vuelve a desplazar hacia atrás, aunque antes habrá que rotar el vector de traslación.

En caso de una rotación secuencial alrededor de varios ejes, naturalmente se han emplear las coordenadas de la rotación anteriormente calculada, como coordena-

das fuente en la siguiente. Si siempre se emplean las coordenadas mundiales como fuente, se deberían obtener resultados bastante extraños, que no tienen mucho que ver con el mundo a representar.

Justamente la rotación con sus muchos cálculos de senos y cosenos representa un caso de aplicación muy apropiado para el cálculo con tablas. Si se quisieran programar todos estos cálculos con las funciones habituales de Pascal, nunca se crearía un movimiento fluido. De modo que todos los valores de seno y coseno se obtienen de una (misma) tabla y se multiplican adecuadamente con las coordenadas en función de la regla matemática correspondiente.

Además de las traslaciones puras, las llamadas transformaciones básicamente no son conmutativas, es decir su orden no es indiferente. Si por ejemplo se rota un punto situado en el eje x (1/0/0) en 90 grados alrededor del eje x, después en el mismo ángulo alrededor del eje z, el resultado se encuentra en el eje y. En un orden inverso se encuentra en el eje z. Por ello se ha de fijar un orden uniforme según el cual se rota. Preferiblemente primero alrededor del eje x, después el y y finalmente el z.

También en el caso de transformaciones combinadas se ha de tener en cuenta el orden. En caso de una traslación con subsiguiente rotación seguramente se obtendrá otro punto que si se hace al revés, por ello también aquí se debería determinar el orden. En este caso lo más adecuado es la secuencia traslación y rotación, ya que los valores de traslación se refieren a las conocidas coordenadas mundiales y no a sus imágenes rotadas. Así que en resumen se puede decir que la siguiente secuencia es la más adecuada:

1. Traslación
2. Rotación (x, y, después z)
3. Proyección en la pantalla

8.4 Modelos de alambre

La posibilidad más simple de obtener en pantalla objetos tridimensionales la representan los modelos de alambre. Aquí sólo se representan las aristas del cuerpo correspondiente, y no sus caras. A causa de ello el cuerpo aparece transparente y uno no se ha de ocupar de la supresión de posibles caras ocultas.

Este modelo utiliza el hecho de que tanto en la proyección paralela como en la perspectiva de punto de fuga descrita, los gráficos aparecen como rectas tridimensionales en la pantalla y no como curvas. Gracias a este hecho se pueden limitar las transformaciones a los vértices, y no se ha de desplazar, rotar y proyec-

tar cada punto de la arista. Si ahora se unen estos vértices calculados, se obtiene una imagen real del cuerpo - un modelo de alambre.

La parte más importante de este modelo es sin duda el algoritmo de líneas. De él depende en gran parte la velocidad de representación posterior, ya que las transformaciones en sí apenas necesitan tiempo de cálculo. Uno de los procedimientos más rápidos actualmente es el algoritmo Bresenham, que también en este capítulo encontrará su aplicación.

En cualquier libro de gráficos se pueden encontrar las bases matemáticas de este algoritmo, así como en libros genéricos sobre el PC y esporádicamente en algunas revistas. Por ello lo obviaremos en este lugar, y nos limitaremos a su principio de funcionamiento básico.

El algoritmo se limita a las pendientes entre 0 y 1 (0 hasta 45 grados). Durante el dibujo de la línea ya sólo se ha de decidir para cada punto si se encuentra exactamente a la derecha del anterior o justo encima de él. No son posibles otros puntos. La decisión de sobre cuál de los dos puntos es el próximo, depende del método de coma fija anteriormente mostrado. Una variable (*Dist*, guardada en BP) se incrementa en *Add_1* (en *S1*) o *Add_2* (en *D1*) dependiendo del último paso (a la derecha o arriba a la derecha). En el siguiente punto la decisión depende de si *Dist* es positivo o negativo.

La limitación a pendientes entre 0 y 1 se puede eliminar fácilmente. Las pendientes entre 1 e infinito (45 hasta 90 grados) se obtienen permutando x e y, y las pendientes negativas mediante la inversión de la dirección de procesamiento. El procedimiento exacto se puede obtener del siguiente listado comentado:

```
.286
b equ byte ptr
w equ word ptr

data segment
  extrn vpage:word                ;página de pantalla actual
data ends

putpixel macro                    ;fija píxeles en ax/bx
  pusha
  xchg ax,bx                      ;permutar x e y
  push ax                         ;guardar y para después
  mov cx,bx                       ;obtener x
  and cx,3                        ;enmascarar plano
  mov ax,1                        ;y fijar bit adecuado
  shl ax,cl
```

```
  mov ah,2                    ;TS Registro 2
  xchg ah,al
  mov dx,3c4h
  out dx,ax

  pop cx                      ;obtener y
  mov ax,80d                  ;obtener offset de líneas
  mul cx
  shr bx,2                    ;añadir offset de columna
  add bx,ax
  add bx,vpage                ;escribir en página actual
  mov b es:[bx],3             ;y fijar color

  popa
endm

code segment public
assume cs:code,ds:data

public bline
bline proc    near
  push bp
  push ax                     ;guardar x0
  push bx                     ;e y0
  mov bx,4340h                ;prepara automodificación
  sub cx,ax                   ;calcular deltax
  jns deltax_ok               ;negativo?
  neg cx                      ;si, invertir signo deltax
  mov bl,48h                  ;y dec ax en vez de inc ax
deltax_ok:
  mov bp,sp                   ;direccionamiento de y1 en la pila
  sub dx,ss:[bp]              ;calcular deltay
  jns deltay_ok               ;¿negativo?
  neg dx                      ;si, invertir signo deltay
  mov bh,4bh                  ;y dec bx en vez de inc bx
deltay_ok:
  mov si,dx                   ;deltay y
  or si,cx                    ;deltax = 0 ?
  jne ok
  add sp,6                    ;entonces ax, bx y bp de pila y fin
  ret
ok:
  mov w cs:dist_pos,bx        ;escribir dec/inc ax/bx a destino
  cmp cx,dx                   ;deltax >= deltay ?
  jge deltax_gross
  xchg cx,dx                  ;no, permitar deltax y deltay
  mov bl,90h                  ;y no hacer inc ax
  jmp constantes
deltax_gross:
  mov bh,90h                  ;si no, no hacer inc bx
```

```
constantes:
  mov w cs:dist_neg,bx          ;escribir dec/inc ax/bx a destino
  shl dx,1                      ;determinar Add_2
  mov di,dx                     ;y guardar en di
  sub dx,cx                     ;determinar dist. inicial
  mov bp,dx                     ;y guardar en bp
  mov si,bp                     ;determinar Add_1
  sub si,cx                     ;y guardar en si
  mov ax,0a000h                 ;cargar segmento VGA
  mov es,ax
  pop bx                        ;recuperar valores de x0 y y0
  pop ax
loop_p:
  putpixel                      ;fijar punto
  or bp,bp                      ;Dist positiva?
  jns dist_pos
dist_neg:
  inc ax                        ;x sigue (evtl automodificación)
  inc bx                        ;y sigue (evtl automodificación)
  add bp,di                     ;actualizar dist
  loop loop_p                   ;siguiente punto
  jmp listo                     ;después listo
dist_pos:
  inc ax                        ;x sigue (evtl automodificación)
  inc bx                        ;y sigue (evtl automodificación)
  add bp,si                     ;actualizar dist
  loop loop_p                   ;siguiente punto
listo:
  pop bp
  ret
bline endp
code ends
end
```

Lo importante para incluirlo en el programa terminado es el paso de parámetros de este procedimiento. Dibuja una línea en el modo X desde el punto con las coordenadas (*AX/BX*) hasta el punto (*CX/DX*). Se tiene en cuenta la página actual (offset en *vpage*).

El paso a la tercera dimensión nos lleva hasta la Unit VAR_3D.PAS, que contiene las variables globales más importantes, cuyo significado se explica en los siguientes apartados:

```
Unit Var_3d;

Interface
Uses Tools;
Const Txt_Numero=5;             {nº de texturas empleadas}
      Txt_tamanyo:              {tamaños de las texturas}
```

```
        Array[0..Txt_Numero-1] of Word=
        ($0a0a,$0a0a,$0a0a,$0a0a,$0a0a);

Var  vz:Word;                    {desplazamiento en la pantalla}
     rotx,                       {ángulo rot.}
     roty,
     rotz:word;                  {pasos de 3 grados}
     fl_sort:Boolean;            {¿ordenar sup.?}
     Rellenar:Boolean;           {true: rellenar / false:líneas}
     fl_dorso:Boolean;           {suprimir caras ocultas?}
     Texture:Boolean;            {emplear texturas?}
     lightsrc:Boolean;           {emplear fuente de luz?}
     Vidrio:Boolean;             {superf. vidrio?}

     Txt_Datos:Array[0..Txt_Numero-1] of Pointer;
                                 {posición de las texturas en memoria}
     Txt_Offs:Array[0..Txt_Numero-1] of Word;
                                 {Offset en la imagen de textura}
     Txt_Pic:Pointer;            {puntero a imagen de textura}

     Seno:Array[0..149] of Word; {tabla senos para rotaciones}

Implementation
Begin
  Sin_Gen(Seno,120,16384,0);
  Move(Seno[0],Seno[120],60);
End.
```

Esta Unit es empleada por todos los programas 3D. Para comenzar inicializa la tabla de senos que se empleará en rotaciones. A continuación se añade el primer cuarto (30 entradas = 60 bytes) de esta tabla, al final de la misma. Esto tiene la ventaja de que se pueden obtener tanto los valores de senos (0 con seno[0]) así como los de cosenos (0 con seno[30]). El programa 3D_DRAHT.PAS emplea el algoritmo Bresenham, que le indica al módulo de ensamblador que dibuje modelos de alambre borrando la variable global *Rellenar*:

```
Uses Crt,ModeXLib,var_3d;

Const
    worldlen=8*3;                {Array de puntos}
    Worldconst:Array[0..worldlen-1] of Integer =
    (-200,-200,-200,
    -200,-200,200,
    -200,200,-200,
    -200,200,200,
    200,-200,-200,
    200,-200,200,
    200,200,-200,
```

```
    200,200,200);
    surfclen=38;                {Array de superficies}
    surfcconst:Array[0..surfclen-1] of Word=
    (0,4, 0,2,6,4,
     0,4, 0,1,3,2,
     0,4, 4,6,7,5,
     0,4, 1,5,7,3,
     0,4, 2,3,7,6,
     0,4, 0,4,5,1,0,0);

Var
    i,j:Word;


procedure drawworld;external;  {dibuja el mundo en la pág. de panta-
lla actual}
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l wurzel.obj}

Begin
  vz:=1000;                     {cuerpo está en mil u. profundidad}
  vpage:=0;                     {comenzar con página 0}
  init_modex;                   {activar ModeX}
  rotx:=0;                      {valores iniciales de rotación}
  roty:=0;
  rotz:=0;
  Fuellen:=false;               {apagar relleno de superficies}
  fl_sort:=false;               {apgar ordenación de superf.}
  fl_ruecken:=false;            {apagar supresion de caras ocultas}
  Glas:=false;                  {apagar superficies de vidrio}
  repeat
    clrx($0f);                  {borrar pantalla}
    DrawWorld;                  {dibujar mundo}
    switch;                     {pasar a imagen terminada}
    WaitRetrace;                {esperar siguiente retrazado}
    Inc(rotx);                  {rotar ... }
    If rotx=120 Then rotx:=0;
    Inc(rotz);
    If rotz=120 Then rotz:=0;
    inc(roty);
    if roty=120 Then roty:=0;
  Until KeyPressed;             { ... hasta tecla}
  TextMode(3);
End.
```

Al principio de este programa se encuentran, al igual que en los siguientes programas, las definiciones del mundo y de las superficies. El mundo sólo contiene las coordenadas de los vértices, que simplemente se indican en el orden

x, y, z para cada punto. La relación entre estos puntos es creada por la definición de la superficie. En este array se determinan las propiedades de las diferentes superficies.

El primer *Word* de cada definición de superficie contiene la estructura de ésta y que sólo importará en apartados posteriores, de modo que ahora vale 0. A continuación vienen el número de vértices de esta cara y el número de vértices en sí, comenzando su enumeración en *0*. El primer rectángulo se compone por ello de los vértices *0, 2, 6, 4*, lo que corresponde a las coordenadas (-200/-200/-200), (-200,200,-200), (200,200,-200), (200,-200,-200). El final de la definición de superficie lo han de formar dos ceros, ya que sino no es reconocida por el procedimiento de representación.

En el programa principal se determina primero la profundidad global vz del objeto. Si se reduce este valor, el objeto se acerca. Las variables *rotx*, *roty* y *rotz* indican el ángulo de rotación del objeto en pasos de 3 grados. Es decir, una rotación de 15 grados queda determinada por un valor de 5.

La elección del modo gráfico recayó en el modo X. Este tiene alguna ligera desventaja de velocidad en el direccionamiento de píxeles individuales, pero puede ofrecer la gran ventaja de las dos páginas de pantalla. Esto es muy importante en las texturas, ya que aquí la actualización de la pantalla dura algo más que un período de retrazado.

A continuación se fijan las llamadas variables globales de control. En este caso se desactivan el relleno de superficies (*Rellenar*), la ordenación (*fl_sort*), la supresión de caras ocultas (*fl_oculto*) y las superficies de vidrio (*vidrio*). Todas estas características sólo se emplearán en los siguientes apartados.

El bucle que sigue a continuación es prácticamente igual para todos los programas: mientras no se pulsa ninguna tecla, se borra la pantalla cada vez, se dibuja el mundo, se conmuta a una nueva página de pantalla, y se sigue rotando.

El trabajo principal de este programa lo realiza el módulo de ensamblador 3DASM.ASM, que está completamente en ensamblador por razones de velocidad. No se deje irritar por el largo y complicado código, en los siguientes apartados se explicarán todos los procedimientos y macros paso a paso. Pero como el código fuente no se puede dividir en diferentes partes, a continuación lo tiene al completo:

```
.286
w equ word ptr
b equ byte ptr
surfclen equ 200          ;longitud máx. de def. de superficie
LongPunt equ 4*100        ;longitud del point-array
num_caras equ 30          ;nº máx. de superficies
```

```
num_vertices equ 10                ;nº máx de vértices
data segment                       ;variables externas de Pascal
  extrn vz:word                    ;produndidad total
  extrn rotx:Word                  ;ángulo de rotación
  extrn roty:Word
  extrn rotz:word
  extrn worldconst:dataptr         ;array con puntos
  extrn surfcconst:dataptr         ;array con definiciones de superficie
  extrn lightsrc:word              ;bandera para el sombreado
  extrn fl_sort:word               ;bandera para la ordenación de super-
ficies
  extrn fl_dorso:word              ;bandera para la supresión de caras
ocultas
  extrn Texture:Byte               ;bandera para texturas
  extrn Rellenar:Byte              ;bandera para relleno/modelo alambre

crotx dw 0                         ;ángulo x, y, z como offset al valor
croty dw 0                         ;senoidal correspondiente
crotz dw 0

rotx_x dw 0                         ;x,y,z a rot. x
rotx_y dw 0
rotx_z dw 0
roty_x dw 0                         ;a rot. y
roty_y dw 0
roty_z dw 0
rotz_x dw 0                        ;a rot. z, definitivo
rotz_y dw 0
rotz_z dw 0

startpoly dw 0                     ;comienzo de def. de la sup. actual

Puntos      dw LongPunt dup (0) ;acoge coordenadas a punteros
Puntosptr   dw 0                   ;en el array de puntos
Puntos3d    dw LongPunt dup (0) ;acoge coord. 3D (textura)
Medio      dw num_caras*2 dup (0);registro de los valores z medios
Medioptr   dw 0                    ;puntero en el array medio
n           dw 0,0,0,0,0,0         ;vector perpendicular 32-bits
n_cant      dw 0                   ;valor del vector perp.

extrn seno:dataptr

data ends

extrn drawpol:near                 ;dibuja superficie como modelo alambre
extrn fillpol:near                 ;rellena la superficie
extrn raiz:near                    ;calcula la raiz de ax

getdelta macro                     ;calcula ambos vectores de sup.
  mov ax,poly3d[0]                 ;x: vértice origen
  mov delta2[0],ax                 ;guardar en delta2
```

```
  sub ax,poly3d[8]              ;crear diferencia con el primer punto
  mov delta1[0],ax              ;y delta1 listo

  mov ax,poly3d[2]              ;y: vértice origen
  mov delta2[2],ax              ;guardar en delta2
  sub ax,poly3d[10d]            ;crear diferencia con el primer punto
  mov delta1[2],ax              ;y delta1 listo

  mov ax,poly3d[4]              ;z: vértice origen
  mov delta2[4],ax              ;guardar en delta2
  sub ax,poly3d[12d]            ;crear diferencia con el primer punto
  mov delta1[4],ax              ;y delta1 listo

  mov bp,polyn                  ;seleccionar último punto
  dec bp
  shl bp,3                      ;a 8 bytes
  mov ax,poly3d[bp]             ;obtener x
  sub delta2[0],ax              ;formar resta
  mov ax,poly3d[bp+2]           ;obtener y
  sub delta2[2],ax              ;formar resta
  mov ax,poly3d[bp+4]           ;obtener z
  sub delta2[4],ax              ;formar resta
endm

FijaCoord macro fuente,offst    ;fija coord. de pantalla terminadas
.386
  mov ax,fuente                 ;proyectar coordenada
  cwd
  shld dx,ax,7
  shl ax,7
  idiv cx
   add ax,offst                 ;centro de pantalla es 0/0/0
  mov bx,Puntosptr              ;guardar en array de puntos
  mov Puntos[bx],ax
  add Puntosptr,2               ;avanzar punto de array
endm

z2cx macro tabofs               ;pasa coord. z a cx
  mov cx,tabofs + 4
  add cx,vz                     ;z-translation
  mov bx,Medioptr               ;guardar en el array medio
  add Medio[bx],cx
endm

xrot macro ZCoord,QCoord        ;rota QCoord en x, guarda en ZCoord
.386
  mov bp,crotx                  ;obtener ángulo
  mov bx,[QCoord]
  shl bx,3                      ;x8, para alinear a puntos
```

```
  mov Puntosptr,bx

  sub bx,[QCoord]                ;x6, para alinear a entradas «mundo»
  sub bx,[QCoord]
  add bx,offset worldconst       ;fijar a mundo
  mov ax,[bx]                    ;obtener x
  mov ZCoord,ax                  ;y fijar sin cambios

  mov ax,[bx+2]                  ;obtener y
  imul w ds:[bp+60d]             ;*cos rotx
  shrd ax,dx,14d
  mov cx,ax                      ;guardar en cx
  mov ax,[bx+4]                  ;obtener z
  imul w ds:[bp]                 ;*-sin rotx
  shrd ax,dx,14d
  sub cx,ax
  mov ZCoord+2,cx                ;valor y listo y fijar

  mov ax,[bx+2]                  ;obtener y
  imul w ds:[bp]                 ;*sin rotx
  shrd ax,dx,14d
  mov cx,ax                      ;guardar en cx
  mov ax,[bx+4]                  ;obtener z
  imul w ds:[bp+60d]             ;*cos rotx
  shrd ax,dx,14d
  add cx,ax
  mov ZCoord+4,cx
endm

yrot macro ZCoord,QCoord         ;rota QCoord en y, guarda en ZCoord
  mov bp,croty                   ;obtener ángulo
  mov ax,QCoord+2                ;obtener y
  mov ZCoord+2,ax                ;y fijar sin cambios

  mov ax,QCoord                  ;obtener x
  imul w ds:[bp+60d]             ;*cos roty
  shrd ax,dx,14d
  mov cx,ax                      ;guardar en cx
  mov ax,QCoord+4                ;obtener z
  imul w ds:[bp]                 ;*sin roty
  shrd ax,dx,14d
  add cx,ax
  mov ZCoord,cx                  ;valor x listo y fijar

  mov ax,QCoord                  ;obtener x
  imul w ds:[bp]                 ;*-sin roty
  shrd ax,dx,14d
  mov cx,ax                      ;guardar en cx
  mov ax,QCoord+4                ;obtener z
```

```
  imul w ds:[bp+60d]            ;*cos roty
  shrd ax,dx,14d
  sub ax,cx
  mov ZCoord+4,ax
endm

zrot macro ZCoord,QCoord        ;rota QCoord en z, guarda en ZCoord
  mov bx,Puntosptr              ;preparar entrada array de puntos 3D

  mov bp,crotz                  ;obtener ángulo
  mov ax,QCoord+4               ;obtener z
  mov ZCoord+4,ax               ;y fijar sin cambios
  mov Puntos3d[bx+4],ax         ;guardar además en el array 3D

  mov ax,QCoord                 ;obtener x
  imul w ds:[bp+60d]            ;*cos rotz
  shrd ax,dx,14d
  mov cx,ax                     ;guardar en cx
  mov ax,QCoord+2               ;obtener y
  imul w ds:[bp]                ;*-sin rotz
  shrd ax,dx,14d
  sub cx,ax
  mov ZCoord,cx                 ;valor x listo y fijar
  mov Puntos3d[bx],cx

  mov ax,QCoord                 ;obtener x
  imul w ds:[bp]                ;*sin rotz
  shrd ax,dx,14d
  mov cx,ax                     ;guardar en cx
  mov ax,QCoord+2               ;obtener y
  imul w ds:[bp+60d]            ;*cos rotz
  shrd ax,dx,14d
  add cx,ax
  mov ZCoord+2,cx
  mov Puntos3d[bx+2],cx

endm

get_normal macro                ;calcula vector perpendicular de sup.
  mov ax,delta1[2]              ;a2*b3
  imul delta2[4]
  shrd ax,dx,4
  mov n[0],ax
  mov ax,delta1[4]              ;a3*b2
  imul delta2[2]
  shrd ax,dx,4
  sub n[0],ax
  mov ax,delta1[4]              ;a3*b1
  imul delta2[0]
```

```
  shrd ax,dx,4
  mov n[2],ax
  mov ax,delta1[0]              ;a1*b3
  imul delta2[4]
  shrd ax,dx,4
  sub n[2],ax
  mov ax,delta1[0]              ;a1*b2
  imul delta2[2]
  shrd ax,dx,4
  mov n[4],ax
  mov ax,delta1[2]
  imul delta2[0]
  shrd ax,dx,4
  sub n[4],ax                   ;producto cruzado (=vector perp.) listo

  mov ax,n[0]                   ;x1 ^ 2
  imul ax
  mov bx,ax
  mov cx,dx
  mov ax,n[2]                   ;+x2 ^ 2
  imul ax
  add bx,ax
  adc cx,dx
  mov ax,n[4]                   ;+x3 ^ 2
  imul ax
  add ax,bx
  adc dx,cx                     ;Suma en dx:ax
  push si
  call raiz                     ;raiz en ax
  pop si
  mov n_cant,ax                 ;valor del vector perp. listo
endm

light macro                     ;determina brillo de una sup.
  mov ax,n[0]
  imul l[0]                     ;vector luz * vector perp.
  mov bx,ax                     ;formar suma en cx:bx
  mov cx,dx
  mov ax,n[2]
  imul l[2]
  add bx,ax
  adc cx,dx
  mov ax,n[4]
  imul l[4]
  add ax,bx                     ;producto escalar listo en dx:ax
  adc dx,cx
  idiv l_cant                   ;dividir por l_cant

  mov bx,n_cant                 ;y por n_cant
```

```
  cwd
  shld dx,ax,5                ;valores de -32 a +32
  shl ax,5d
  mov bp,startpoly            ;preparar direcc. del color de sup.
  idiv bx                     ;división por el divisor
  inc ax
  or ax,ax
  js encarado                 ;si cos α positivo -> apartado de la
luz
  xor ax,ax                   ;así que no hay luz
encarado:
  sub b polycol,al            ;cos<0 -> sumar a color básico
endm

code segment
assume cs:code,ds:data

public drawworld

public linecount
public polycol
public polyn
public poly2d
public poly3d
linecount dw 0
polycol dw 3                  ;color de superficie actual
polyn   dw 0                  ;nº de vértices existentes realmente
poly2d dw num_vertices*4 dup (0) ;vértices del polígono a dibujar
poly3d dw num_vertices*4 dup (0) ;vértices 3D

public Txt_Nr
Txt_Nr dw 0                   ;nº de la textura actual

public delta1,delta2
delta1      dw 0,0,0          ;vectores de plano
delta2      dw 0,0,0

l           dw 11d,11d,11d    ;vector de luz
l_cant      dw 19d            ;valor del vector luz

drawworld proc pascal         ;dibuja mundo tridimensional
  push ds
  push es
  push bp
  lea si,surfcconst           ;superf. se direccionan mediante si
  mov Medioptr,0              ;comenzar con 0 en el array medio
  mov ax,ds:[rotx]            ;obtener ángulo
  shl ax,1                    ;convertir en offset de memoria
  add ax,offset seno
```

```
  mov crotx,ax                  ;y depositar en variable aux.
  mov ax,ds:[roty]              ;igual para y...
  shl ax,1
  add ax,offset seno
  mov croty,ax
  mov ax,ds:[rotz]              ;...y z
  shl ax,1
  add ax,offset seno
  mov crotz,ax
npoly:                          ;bucle polígono
  mov startpoly,si              ;guardar para uso posterior
  add si,2                      ;saltar color
  mov cx,[si]                   ;obtener nº vértices
  mov linecount,cx              ;cargar contador
  inc cx                        ;a causa de sup. cerrada
  mov w polyn,cx                ;guardar en array de puntos
  add si,2                      ;seguir a las coordenadas reales

nline:
  xrot rotx_x,si                ;rotar coord. en x
  yrot roty_x,rotx_x            ;en y
  zrot rotz_x,roty_x            ;y en z
  z2cx rotz_x                   ;obtener inicio z

  FijaCoord rotz_x,160          ;escribir coordenadas
  FijaCoord rotz_y,100

  add si,2                      ;siguiente vértice
  dec linecount                 ;aumentar contador de líneas
  je polyok                     ;todas dibujadas -> fin
  jmp nline                     ;sino siguiente línea

polyok:
  mov bx,Medioptr               ;obtener valor medio:
  mov ax,Medio[bx]              ;obtener suma
  mov cx,polyn
  dec cx
  cwd
  div cx                        ;y dividir por nº de vértices
  mov Medio[bx],ax              ;escribir
  mov ax,startpoly              ;también escribir «nº» de la superfi-
cie
  mov Medio[bx+2],ax
  add Medioptr,4                ;y seguir
  cmp w [si+2],0                ;¿todos los polígonos listos?
  je listo
  jmp npoly

listo:
  cmp b fl_sort,0               ;¿ordenar superficies?
```

```
  je sin_quicksort
  call quicksort pascal,0,bx      ;ordenar campo desde 0 a pos. actual
sin_quicksort:
  mov Medio[bx+4],0               ;colocar fin
  mov ax,cs                       ;fijar segmento destino
  mov es,ax
  xor bx,bx                       ;comenzar con 1ª superficie
npoly_draw:
  lea di,poly2d                   ;destino: poly-array
  mov bp,Medio[bx+2]              ;obtener puntero color y puntos de
sup.
  mov ax,ds:[bp]                  ;obtener y fijar color
  mov polycol,ax

  mov texture,0                   ;suposición: sin textura
  cmp ah,0ffh                     ;Textura?
  jne sin_textura
  mov texture,1                   ;si, fijar
  mov b txt_nr,al                 ;guardar nº

sin_textura:
  mov b lightsrc,0                ;suposición: sin sombreado
  cmp ah,0feh                     ;sombreado?
  jne keine_Lichtfuente
  mov b lightsrc,1                ;si, fijar

keine_Lichtfuente:
  add bp,2                        ;posicionar en cantidad
  mov cx,ds:[bp]                  ;obtener nº de vértices
  mov polyn,cx                    ;escribir en poly-array
npoint:
  add bp,2
  mov si,ds:[bp]                  ;obtener puntero a punto real
  shl si,3                        ;3 entradas WORD!
  add si,offset Puntos            ;y x/y del array de puntos en coord.
poly

  mov ax,[si+Puntos3d-Puntos]     ;obtener x 3D
  mov es:[di+poly3d-poly2d],ax    ;fijar x 3D
  mov ax,[si+Puntos3d-Puntos+2]   ;obtener y 3D
  mov es:[di+poly3d-poly2d+2],ax;fijar y 3D
  mov ax,[si+Puntos3d-Puntos+4]   ;obtener z 3D
  mov es:[di+poly3d-poly2d+4],ax;fijar z 3D

  movsw                           ;fijar coordenadas 2D
  movsw

  add di,4                        ;siguiente entrada Poly2d
  dec cx                          ;¿todos los vértices?
```

```
  jne npoint
  mov bp,polyn                  ;copiar primer vértice al último
  shl bp,3                      ;posicionar en primer punto
  neg bp
  mov ax,es:[di+bp]             ;y copiar
  mov es:[di],ax
  mov ax,es:[di+bp+2]
  mov es:[di+2],ax

  add di,poly3d-poly2d          ;lo mismo para coord. 3D
  mov ax,es:[di+bp]             ;y copiar
  mov es:[di],ax
  mov ax,es:[di+bp+2]
  mov es:[di+2],ax
  mov ax,es:[di+bp+4]
  mov es:[di+4],ax

  cmp rellenar,1                ;¿rellenar superficie?
  jne lines

  getdelta                      ;si, calcular deltal y 2
  cmp b lightsrc,0              ;¿fuente de luz?
  jne sombrear
  jmp sin_luz

sombrear:                       ;si,
  push bx
  get_normal                    ;calcular vector perp.
  light                         ;y brillo
  pop bx

sin_luz:
  inc polyn                     ;aumentar nº de vértices
  call fillpol                  ;dibujar sup.
next:
  add bx,4                      ;siguiente sup.
  cmp Medio[bx],0               ;¿última?
  je _npoly_draw                ;no, seguir
  jmp npoly_draw

lines:
  push bx
  call drawpol                  ;dibujar polígono
  pop bx
  jmp next

_npoly_draw:
```

```
  pop bp                          ;y listo
  pop es
  pop ds
  ret
drawworld endp


public quicksort
quicksort proc pascal unten,oben:word
;ordena array de medios según algoritmo QuickSort

local clave:word
local links:word
  push bx
  mov bx,unten                    ;encontrar centro
  add bx,oben
  shr bx,1
  and bx,not 3                    ;posicionar en bloques de 4
  mov dx,Medio[bx]                ;obtener llave
  mov clave,dx
  mov  ax,unten                   ;inicializ. dcha. e izq. con valores base
  mov si,ax
  mov links,ax
  mov ax,oben
  mov di,ax
  mov dx,clave
izq_cerca:
  cmp Medio[si],dx                ;mayor que clave -> seguir buscando
  jbe toca_izq
  add si,4                        ;posicionar al siguiente
  jmp izq_cerca                   ;y comprobarlo
toca_izq:
  cmp Medio[di],dx                ;menor que clave -> seguir buscando
  jae toca_dcha
  sub di,4                        ;posicionar al siguiente
  jmp toca_izq                    ;y comprobarlo
toca_dcha:
  cmp si,di                       ;izq. <= dcha.?
  jg fin_clave                    ;no -> zona parcial ordenada
  mov eax,dword ptr Medio[si]     ;cambiar valores medios y posiciones
  xchg eax,dword ptr Medio[di]
  mov dword ptr Medio[si],eax

  add si,4                        ;mover puntero
  sub di,4
fin_clave:
  cmp si,di                       ;izq. > dcha., seguir
  jle izq_cerca
```

```
  mov links,si                    ;guardar izq., por recursión
  cmp unten,di                    ;abajo < dcha. -> ordenar parte izq.
  jge rechts_listo
  call quicksort pascal,unten,di;ordenar mitades recursivamente
rechts_listo:
  mov si,links                    ;arriba>izq. -> ordenar parte dcha.
  cmp oben,si
  jle links_listo
  call quicksort pascal,si,oben ;ordenar mitades recursivamente
links_listo:
  pop bx
  ret
quicksort endp

code ends
end
```

El significado central del procedimiento *Drawworld* ya se puede ver en su longitud. Aquí se dibuja el mundo tridimensional completo, dependiendo de los valores actuales de transformación. No se pasan parámetros, y el control se realiza completamente a través de variables globales.

Justamente al principio se convierten los valores actuales de rotación (*rotx, roty, rotz*) en offsets con respecto a su segmento de datos (*crotx, croty, crotz*), de modo que el acceso se podrá realizar posteriormente sin más cálculos. En el bucle que sigue, *npoly*, que se recorre una vez por cada polígono, el registro SI sirve para el direccionamiento dentro del array de superficies (*surfc_const*).

La información de color es saltada aquí, y se lee directamente el número de vértices del polígono del array. Este valor se guarda en el contador *LineCount*, que a continuación indica la cantidad de vértices a calcular. Después el procedimiento entra en el bucle *nline*, que rota los puntos uno detrás de otro, proyectándolos en las coordenadas de pantalla. Las rotaciones (macros *xrot, yrot, zrot*) pasan el resultado de su trabajo al sucesor, de modo que las coordenadas originales son rotadas en secuencia alrededor de los ejes *x*, *y* y *z*.

La estructura de las tres macros de rotación es muy similar: *BP* contiene el ángulo en forma de offset, de modo que el valor senoidal correspondiente se puede leer con facilidad en la tabla mediante indexación con *BP*. El coseno se obtiene de forma similar, sumando 60 (bytes, es decir 30 entradas) a *BP*. Para poder acceder posteriormente al array tridimensional de puntos (con entradas de 8 bytes), el número del vértice actual se ha de multiplicar con ocho. Este índice se guarda en la variable *PuntosPtr*. Después de una doble resta, ya no queda nada en contra de un direccionamiento de los vértices de 6 bytes.

La secuencia subsiguiente es muy similar para todas las macros. Para cada una de las tres coordenadas se calcula una proyección en función de la matriz de rotación. En el caso de la rotación *x*, por ejemplo, se puede adoptar la coordenada x sin modificaciones. *Y* y *z* se han de determinar según las conocidas fórmulas $y' = y * cos\alpha - z * sin\alpha$ y $z' = y * sin\alpha + z * cos\alpha$.

Las otras dos macros son idénticas excepto en una matriz, y sólo la rotación z coloca las coordenadas terminadas en el array 3D *Puntos3D*, que en este lugar aún no tiene ninguna importancia.

La macro que viene a continuación *z2cx* en este momento sólo tiene la tarea que corresponde a su nombre. Carga la coordenada z rotada en el registro *CX*, teniendo en cuenta la profundidad total *VZ*.

Ahora ya se dispone de las coordenadas del vértice procesado en su forma tridimensional definitiva y sólo resta proyectarlas en la pantalla bidimensional. Esta tarea la realiza la macro *SetCoord*. Proyecta las coordenadas pasadas en *fuente* (por una parte *rotz_x* y por la otra *rotz_y*) y desplaza simultáneamente la imagen acabada al centro de la pantalla, sumando los offsets pasados como segundo parámetro.

La multiplicación con la distancia ocular *a* se obtiene mediante un desplazamiento a la izquierda en 7 bits, lo que corresponde a una multiplicación por *128*. En este lugar los 16 bits del registro *AX* comienzan a ser un poco justos en el caso de coordenadas algo grandes, de modo que se emplea la combinación *DX:AX*. Después de la división entre la coordenada *z* (guardada en *CX* gracias a *z2cx*) se suma el offset y se guarda la coordenada 2D terminada en el lugar correspondiente al número de vértice del array de puntos. Después de cada escritura en este array se avanza el puntero en dos bytes, para poder depositar en la misma macro las coordenadas *x* e *y*. Después de estas macros se cierra el bucle *nline* mediante el direccionamiento del siguiente vértice (incrementando *SI*) y el decremento del contador de líneas.

Una vez calculados todos los vértices, se obtiene la profundidad media del polígono a partir de la etiqueta *polyok*, pero luego más sobre ello. En este lugar sólo es importante que en el array *medio*, la segunda palabra permite una identificación del polígono correspondiente por medio de su dirección inicial (guardada en *Startpoly*). Para los modelos de alambre nos vuelve a interesar la instrucción *cmp* al finalizar el bucle *npoly*. Aquí, en el caso de un cero en la cantidad de vértices, se finaliza el bucle y se salta a la etiqueta *listo*.

En modelos posteriores se realiza en este lugar una ordenación de las superficies, pero aquí nos la saltamos, a causa de la variable *fl_Sort*, que está borrada. Se continúa en la etiqueta *sin_Quicksort*. Aquí se coloca un final en el array *medio* mediante

un cero, y después se dibuja cada polígono individualmente en el bucle *npoly_draw*. Para pasar las coordenadas a la rutina *DrawPoly*, que se encarga del dibujo en sí, se emplea otro array con el nombre *Poly2d*. Cada vez se obtiene la identificación del polígono del array medio y mediante ella se carga la información de color del campo *SurfcConst*, para guardarla en *polycol*.

Los dos siguientes bloques de instrucciones sirven para la activación de las rutinas de textura y luces, que en este caso no se necesitan. Se continua en la etiqueta *sin_luces* con la lectura del número de vértices del array *SurfcConst*, que se guarda en la variable *polyn*. El siguiente bucle *npoint*, que se recorre para cada vértice, lee primero el número del vértice actual y direcciona mediante él las coordenadas bidimensionales del mismo en el array de puntos. Las dos instrucciones *movsw* copian ahora las coordenadas de este array al campo *Poly2D*, cuyo puntero *DI* se ha de incrementar, para finalizar el bucle.

Ahora sólo falta copiar el primer vértice al último, para obtener un trazo cerrado. Para ello se emplea el siguiente truco: si se multiplica el número de vértices (*polyn*) con ocho (nº de bytes por vértice), se obtiene la longitud total del array *Poly2D* en bytes. Esta longitud simplemente se resta al leer la posición actual (por ello *neg bp*) y se copia a la palabra que hay ahí.

Como este programa emplea modelos de alambre (*rellenar = false = 0*), se salta a la etiqueta *lines*, que simplemente llama el procedimiento *DrawPoly* del módulo POLY.ASM y con ello representa en pantalla el polígono como modelo de alambre. Después se direcciona el siguiente polígono en el array medio y se salta al inicio del bucle *npoly_Draw*. El dibujo del polígono se realiza en el módulo POLY.ASM, en este caso por el procedimiento *DrawPoly*:

```
public drawpol
;dibuja modelo de alambre de la sup. en Poly2d
drawpol proc near
  push es
  pusha
  xor si,si                    ;indice a la primera entrada
  mov bp,polyn                 ;obtener nº de vértices
@nline:
  mov ax,poly2d[si]            ;obtener coordenadas de la tabla
  mov bx,poly2d[si+2]
  mov cx,poly2d[si+8]
  mov dx,poly2d[si+10d]
  push bp
  push si
  call bline                   ;dibujar línea
  pop si
```

```
    pop bp
    add si,8                ;siguiente línea
    dec bp                  ;decrementar nº
    jne @nline
    popa
    pop es
    ret
drawpol endp
```

Aquí se trata claramente de un procedimiento de representación más sencillo. Simplemente se leen las coordenadas de inicio y final para cada punto del array *Poly2D* y se llama al procedimiento *bline*. *SI* sirve como índice en el array y *BP* como contador para la cantidad de líneas a dibujar.

8.5 Vea a través - cuerpos transparentes

Los objetos mostrados hasta ahora prácticamente sólo se componen de aristas, por ello tienen muy poco que ver con objetos reáles. El siguiente paso ha de ser el de darle laterales a los cuerpos y con ello masa. Pero aquí es donde se encuentra en seguida uno de los mayores problemas de todos los mundos 3D calculados: las caras ocultas. Si simplemente se van dibujando las diferentes superficies una detrás de otra, tal como están definidas, frecuentemente aparecen caras que habitualmente no serían visibles. El siguiente capítulo se ocupará de la supresión de estas caras. Aquí vamos a tomar otro camino.

En vez de adaptar las imágenes a la realidad, vamos a adaptar la realidad a las imágenes. En un cuerpo de vidrio siempre se pueden ver todas las caras. A pesar del ello, las caras naturalmente también pueden tener color. Incluso es obligatorio si se quiere que generen una imagen. Si ahora hay dos superficies una detrás de otra, los colores se superponen y como resultado se obtiene un color de mezcla algo más oscuro (dos superficies filtran más luz que una sola). En principio se ha de considerar cualquier combinación de superficies, es decir, si la superficie A puede superponerse a la superficie B con cualquier ángulo, ha de existir un color combinado de ambas. La única posibilidad de realizar esto consiste en reservar un bit en la información de color para cada superficie. Y sólo las superficies que no se superponen en ninguna circunstancia pueden emplear el mismo bit, ya que sino no es posible una mezcla.

Ahora no se sobrescribe al color antiguo al representar la superficie, sino que ambos valores se operan con *OR*, se genera un nuevo valor de color. Si la superficie A por ejemplo tiene el color 2 (bit 1 activo) y la superficie B el color 16 (bit 4), el resultado de la combinación de colores es 18 (bits 1 y 4).

Naturalmente la paleta ha de seguir esta estructura particular. Por una parte ha de contener colores puros, y por otra cualquier combinación de bits de estos colores. El mencionado color 18 ha de ser en cualquier caso una mezcla de los colores 2 y 16.

La paleta se ha de preparar en consecuencia al iniciar el programa, y al rellenar los polígonos se emplea la unidad aritmética de la VGA. Esta se puede conmutar al modo *OR* mediante el registro GDC *3* que opera los datos que llegan de la CPU con los que se encuentran en los Latches, antes de escribirlos a la memoria de vídeo. Ahora sólo se han de cargar los Latches antes del acceso de escritura con los valores que se encuentran en la memoria de vídeo, realizando un acceso de lectura a la misma posición de memoria. El siguiente programa 3D_GLAS.PAS se apoya en el módulo 3DASM.ASM, pero al contrario que en el ejemplo anterior se activan otras variables:

```
Uses Crt,ModeXLib,var_3d;

Const
  worldlen=8*3;                    {array de puntos}
  Worldconst:Array[0..worldlen-1] of Integer =
  (-200,-200,-200,
  -200,-200,200,
  -200,200,-200,
  -200,200,200,
  200,-200,-200,
  200,-200,200,
  200,200,-200,
  200,200,200);
    surfclen=38;                   {array de superficies}
    surfcconst:Array[0..surfclen-1] of Word=
    (01,4, 0,2,6,4,
     02,4, 0,1,3,2,
     04,4, 4,6,7,5,
     08,4, 1,5,7,3,
     16,4, 2,3,7,6,
     32,4, 0,4,5,1,0,0);
Var
    i,j:Word;

Procedure Pal_Vidrio;
{prepara la paleta para cuerpos de vidrio}
Begin
  FillChar(Palette[3],765,63); {primero todos los colores a blanco}
  For i:=1 to 255 do Begin     {determinar 255 colores de mezcla}
     If i and 1 = 1 Then Dec(Palette[i*3],16);
     If i and 2 = 2 Then Dec(Palette[i*3+1],16);
```

```
      If i and 4 = 4 Then Dec(Palette[i*3+2],16);
      If i and 8 = 8 Then Begin
                            Dec(Palette[i*3],16);
                            Dec(Palette[i*3+1],16);
                          End;
      If i and 16 = 16 Then Begin
                            Dec(Palette[i*3],16);
                            Dec(Palette[i*3+2],16);
                          End;
      If i and 32 = 32 Then Begin
                            Dec(Palette[i*3+1],16);
                            Dec(Palette[i*3+2],16);
                          End;
    End;
    SetPal;
  End;
  procedure drawworld;external;   {dibuja el mundo en la pág. actual de
  pantalla}
  {$l 3dasm.obj}
  {$l poly.obj}
  {$l bres.obj}
  {$l wurzel.obj}

  Begin
    vz:=1000;                     {cuerpo se encuentra en mil de prof.}
    vpage:=0;                     {comenzar con pág. 0}
    init_modex;                   {activar Modo X}
    Pal_Vidrio;
    rotx:=0;                      {valores inic. para rotación}
    roty:=0;
    rotz:=0;
    Rellenar:=true;               {activar relleno sup.}
    fl_sort:=false;               {ordenación sup. apagado}
    fl_dorso:=false;              {supresión de caras ocultas apagada}
    Vidrio:=true;                 {superficies de vidrio activadas}
    repeat
      clrx($0f);                  {borrar pantalla}
      DrawWorld;                  {dibujar mundo}
      switch;                     {conmutar a imagen acabada}
      WaitRetrace;                {esperar siguiente retrazado}
      Inc(rotx);                  {seguir rotando... }
      If rotx=120 Then rotx:=0;
      Inc(rotz);
      If rotz=120 Then rotz:=0;
      inc(roty);
      if roty=120 Then roty:=0;
    Until KeyPressed;             { ... hasta tecla}
    TextMode(3);
  End.
```

Mediante la activación de la variable *rellenar* se activa el algoritmo de llenado de superficies, además se coloca *Vidrio* en *TRUE*, con lo que el GDC se pasa al modo *OR*. Antes del programa principal, que básicamente se corresponde con el del apartado anterior, se llama el procedimiento *Pal_vidrio*, que prepara la paleta para los cuerpos de vidrio.

Para ello se reduce la parte del rojo en 16 para cada color cuyo bit 0 esté activo. Con ello aparecen un poco más oscuros todos los colores de mezcla en los que participa el color *1*. Esto significa que la componente del rojo se filtra parcialmente, tal y como ocurre en la realidad. Así mismo se procede con los otros 5 bits, que filtran otros colores.

En el módulo 3DASM.ASM el procesamiento es prácticamente el mismo que en los modelos de alambre. Sólo al final del procedimiento *DrawWorld* no se bifurca a causa de la variable *Rellenar* (que está en *TRUE*), sino que se continúa en la etiqueta *sin_luz*. Allí se llama *FillPol* en vez de *DrawPol*, y el bucle se finaliza como anteriormente descrito.

En este punto entra el resto del módulo POLY.ASM. Aquí se genera una superficie de color según un algoritmo de rellenado muy rápido, con las coordenadas indicadas en *Poly2D*.

Los algoritmos de relleno se dividen en dos categorías: por una parte el relleno común, que rellena superficies cualesquiera ya dibujadas y que frecuentemente se emplea en los programas de dibujo. Por otra parte tenemos el relleno de un polígono definido por coordenadas. Este último algoritmo es incomparablemente más rápido para estos propósitos.

Fundamentalmente, el método aquí descrito se basa en el dibujo de líneas. Para ello se van detectando los bordes derecho e izquierdo del polígono partiendo del punto con la coordenada y menor, hasta que se alcanza la coordenada *y* mayor. Las líneas que delimitan al polígono sólo se calculan y no se dibujan. Cuando se ha avanzado una línea de esta forma, se puede tirar una línea horizontal entre los puntos derecho e izquierdo que se han calculado. Aquí se aprovecha que en el modo X las líneas horizontales se pueden dibujar con una gran velocidad.

Así que la rutina de relleno ha de estar calculando constantemente líneas en el borde derecho e izquierdo del polígono. Si la línea se ha terminado de dibujar, se comienza la siguiente, cuyo punto inicial corresponde al último vértice. El procedimiento *FillPol* muestra cómo se puede traducir a la práctica este planteamiento teórico. Aparte del procedimiento *DrawPol* ya mencionado, se emplea el código completo del módulo POLY.ASM para el algoritmo de relleno.

```
.286
w equ word ptr
b equ byte ptr

include texture.inc                ;implementar macros de textura

setnewlinel macro                  ;¡Aquí sólo emplear ax y bx !
local dylpos,dxlpos,tam_dxl,macro_lista
  mov bx,4043h                     ;para inc ax (en bh) e inc bx (en bl)
  mov bp,izquierda
  mov ax,poly2d[bp+8]              ;guardar coordenadas destino
  mov xl1,ax
  mov ax,poly2d[bp+10d]
  mov yl1,ax

  mov ax,poly2d[bp]                ;izq. inicio x/y en var. global
  mov xl0,ax
  sub ax,xl1                       ;formar deltax
  inc xl1                          ;para la condición de interrupción
  neg ax                           ;xl1-xl0
  jns dxlpos                       ;dxl negativo?
  neg ax                           ;formar valor
  mov bh,48h                       ;código para dec ax (dec xl0)
  sub xl1,2                        ;ampliación de la coordenada destino
a negativo
dxlpos:
  mov dxl,ax                       ;y guardar global
  mov incflagl,ax                  ;guardar en la bandera de incremento
  mov ax,poly2d[bp+2]
  mov yl0,ax
  sub ax,yl1                       ;formar |delta y|
  inc yl1                          ;para la condición de interrupción
  neg ax
  jns dylpos                       ;negativo?
  neg ax                           ;formar valor
  mov bl,4bh                       ;código para dec bx (dec yl1)
  sub yl1,2                        ;ampliación coordenada destino a neg.
dylpos:
  mov dyl,ax                       ;y guardar global
  cmp dxl,ax                       ;dx < dy
  jae tam_dxl
  neg incflagl                     ;cambio signo para bandera de incr.
tam_dxl:
  mov cs:byte ptr incxl,bh         ;realizar automodificación
  mov cs:byte ptr incyl,bl

  cmp texture,1                    ;se necesitan texturas?
  jne macro_lista                  ;no, saltar

  txt_makevarl                     ;si no calcular variables de textura
```

```
macro_lista:
  mov ax,xl0                        ;emplear registro como variable
contadores
  mov bx,yl0
  mov si,incflagl
endm

setnewliner macro                   ;emplear sólo cx y dx !
local dyrpos,dxrpos,dxrgross,macro_lista
  mov cx,4142h                      ;para inc cx (en ch) e inc dx (en cl)
  mov bp,derecha
  mov dx,poly2d[bp]                 ;obtener coordenadas destino
  mov xr1,dx
  mov dx,poly2d[bp+2]
  mov yr1,dx
  mov dx,poly2d[bp+8]               ;dcha. x/y en var glob.
  mov xr0,dx
  sub dx,xr1                        ;formar |delta x|
  inc xr1                           ;para la condición de interrupción
  neg dx
  jns dxrpos                        ;negativo?
  neg dx                            ;formar valor
  mov ch,49h                        ;código para dec cx
  sub xr1,2                         ;ampliación coordenada destino a
nega.
dxrpos:
  mov dxr,dx                        ;guardar en variable global
  mov incflagr,dx
  mov dx,poly2d[bp+10d]             ;formar |delta y|
  mov yr0,dx
  sub dx,yr1
  inc yr1                           ;para la condición de interrupción
  neg dx
  jns dyrpos                        ;¿negativo?
  neg dx                            ;formar valor
  mov cl,4ah                        ;código para dec dx
  sub yr1,2                         ;ampliación de la coordenada desti-
no a negativo
dyrpos:
  mov dyr,dx                        ;y guardar en var. glob.
  cmp dxr,dx                        ;dx < dy ?
  jae dxrgross
  neg incflagr                      ;cambio signo para bandera incremento
dxrgross:
  mov cs:byte ptr incxr,ch          ;automodificación
  mov cs:byte ptr incyr,cl

  cmp texture,                      ;se necesitan texturas?
  jne macro_lista                   ;no, saltar
```

```
  txt_makevarr                ;sino calcular variables de textura
macro_lista:
  mov cx,xr0                  ;cargar registro
  mov dx,yr0
  mov di,incflagr
endm

data segment public
  extrn vpage:word            ;página de pantalla actual
  extrn fl_dorso              ;flag para supresión de caras ocultas
  extrn vidrio:Byte;          ;bandera para superficies de vidrio

;Variables de textura:
  extrn Texture:Byte          ;se necesita textura?
  extrn Txt_Daten:DataPtr     ;Array con punteros a datos gráficos
  extrn Txt_Offs:DataPtr      ;Array con Offsets en imagen textura
  extrn Txt_tamanyo:DataPtr   ;Array con indicaciones de tamaño

d_x  dd 0                     ;coordenada x relativa
d_y dd 0                      ;coordenada y relativa
D    dd 0                     ;determinante principal
Columna1 dd 0                 ;compon. de determinante principal
        dd 0
Columna2 dd 0
        dd 0
fila_superior dw 0            ;¿qué coordenadas se emplean?
fila_inferior dw 0

xl_3d dd 0                    ;valores para coord. 3D al rellenar
yl_3d dd 0
zl_3d dd 0
xr_3d dd 0
yr_3d dd 0
zr_3d dd 0

inc_xl dd 0                   ;valores para sumar a ellos
inc_yl dd 0
inc_zl dd 0
inc_xr dd 0
inc_yr dd 0
inc_zr dd 0

;Variables para algoritmo de relleno
inc_punto   dw 0              ;mantener en dx durante búsqueda
inc_y       dw 0              ;mantener en bx durante búsqueda

izquierda        dw 0         ;punto del lado izq.
derecha      dw 0             ;punto del lado dcho.
```

```
xl0  dw 0                    ;valores para puntos inicial y final
yl0 dw 0
xl1 dw 0
yl1 dw 0
xr0 dw 0                     ;valores para dcha.
yr0 dw 0
xr1 dw 0
yr1 dw 0
dxl dw 0                     ;Delta X / Y para ambos lados
dyl dw 0
dxr dw 0
dyr dw 0
incflagl dw 0                ;banderas, cuándo se ha de incrementar y
incflagr dw 0                ;una especie de «pendiente»
data ends
code segment public
assume cs:code,ds:data
extrn polycol:word           ;color de superficie
extrn polyn:word             ;nº vértices
extrn poly2d:word            ;Array con coord. 2D
extrn poly3d:word            ;Array con coord. 3D
extrn delta1,delta2:word     ;vectores de plano
extrn bline:near             ;dibuja línea

lambda1 dd 0                 ;coord. afines
lambda2 dd 0

inc_lambda1 dd 0             ;amplitud de paso
inc_lambda2 dd 0

plane dw 0002h               ;plane a fijar actual
x0 dw 0                      ;coordenadas para línea
y0 dw 0
x1 dw 0
zz dw 0                      ;puntos a dibujar

extrn Txt_Nr:Word            ;nº de la textura a dibujar

public drawpol
;dibuja modelo de alambre de la sup. en Poly2d
drawpol proc near
  push es
  pusha
  xor si,si                  ;índice a la primera entrada
  mov bp,polyn               ;obtener nº de vértices
@nline:
  mov ax,poly2d[si]          ;obtener coordenadas de la tabla
```

```
  mov bx,poly2d[si+2]
  mov cx,poly2d[si+8]
  mov dx,poly2d[si+10d]
  push bp
  push si
  call bline                    ;dibujar línea
  pop si
  pop bp
  add si,8                      ;siguiente línea
  dec bp                        ;decrementar nº
  jne @nline
  popa
  pop es
  ret
drawpol endp

hline proc near                 ;dibuja línea horiz. ax/bx -> cx/bx
  pusha
  push es
  mov x0,ax                     ;guardar coord. para después
  mov y0,bx
  mov x1,cx
  sub cx,ax                     ;calcular nº de puntos a dibujar
  jne zzok
  inc cx

zzok:
  mov zz,cx

  cmp vidrio,1                  ;superficies de vidrio
  jne Solid1
  push ax                       ;si, modo GDC: OR
  mov dx,3ceh
  mov ax,1003h                  ;Registro 3: Function Select
  out dx,ax
  pop ax

Solid1:
  mov dx,3c4h                   ;Timing Sequencer-Port
  mov di,0a000h
  mov es,di                     ;seleccionar segmento CGA
  mov di,ax                     ;calcular offset
  shr di,2                      ;(x div 4) + y*80
  add di,vpage                  ;sumar página actual
  mov bx,y0
  imul bx,80d
  add di,bx                     ;ahora en di
  cmp zz,4
  jl sin_centro                 ;<4 puntos -> sin bloques de 4
```

```
  and ax,11b                    ;dos bits inferiores son importantes
  je centro                     ;si 0 colocar bloques de 4
sin_centro:
  mov bx,0f02h                  ;si sin_shift, usar esta máscara
  mov cx,zz                     ;fijar nº de puntos en máscara
  cmp cx,20h                    ;desde de 20h, 386 desplaza de nuevo
  jae sin_shift
  mov bx,0102h                  ;preparar máscara
  shl bh,cl                     ;nº puntos=nº de bits a activar
  dec bh
  and bh,0fh
sin_shift:
  mov cx,ax                     ;desplazar según plane inicial
  and cl,3
  shl bh,cl
  mov ax,bx                     ;y máscara lista
  sub zz,4                      ;descontar puntos a dibujar
  add zz,cx
start:
  out dx,ax                     ;activar máscara escritura calculada
  mov al,b polycol              ;obtener color

  mov ah,es:[di]                ;cargar latches, sólo cuerpo vidrio
  stosb                         ;fijar
centro:
  cmp zz,4
  jl final                      ;si no hay bloques de 4 -> terminar

  mov ax,0f02h                  ;seleccionar todos los planes
  out dx,ax                     ;(zz div 4) fijar bloques de 4
  mov cx,zz
  shr cx,2
  mov al,b polycol

  cmp vidrio,1                  ;¿cuerpo de vidrio?
  jne Solid

@lp:
  mov ah,es:[di]                ;cargar latches, sólo para cuerpo de
vidrio
  stosb                         ;y escribir
  dec cx
  jne @lp
  jmp final

Solid:
  rep stosb                     ;dibujar parte central

final:
```

```
  mov cx,x1                    ;activar restantes píxeles
  and cx,3h
  dec zz
  js hline_lista               ;si no queda nada -> fin
  mov ax,0102h
  shl ah,cl                    ;crear máscara
  dec ah
  out dx,ax
  mov al,b polycol             ;obtener color

  mov ah,es:[di]               ;cargar latches, sólo cuerpo vidrio

  stosb                        ;y dibujar puntos
hline_lista:
  mov dx,3ceh                  ;modo GDC a MOVE
  mov ax,0003h
  out dx,ax
  pop es
  popa
  ret
  hline endp

  txt_hline                    ;Macro contiene el proc.
«hline_texture»

public fillpol
fillpol proc near              ;rellena polígono en modo X
  push bp
  pusha

  cmp texture,1                ;¿se emplean texturas?
  jne Füllen                   ;no, simplemente rellenar

  txt_DetPrinc                 ;sino calcular determinante principal

Füllen:
  xor si,si                    ;buscar punto mayor, selecc. 1ª entra-
da
  mov cx,polyn                 ;nº vértices
  sub cx,2
  mov bx,0ffffh                ;valor muy alto, aun así por debajo
npoint:
  mov ax,poly2d[si+2]          ;obtener y
  cmp ax,bx                    ;si por debajo de mínimo actual
  ja  no_min
  mov bx,ax                    ;fijar nuevo mínimo
  mov dx,si
no_min:
  add si,8
  dec cx                       ;sig. vértice, sino 0ffffh
```

```
  jns npoint
  mov inc_punto,dx              ;retener en var global
  mov inc_y,bx                  ;búsqueda de punto alto finalizada

  or dx,dx                      ;izq. = 0 ?
  jne dec_valid
  mov bx,polyn                  ;si: al final a la dcha.
  sub bx,2
  shl bx,3
  jmp lr_lista                  ;posicionar
dec_valid:
  mov bx,dx                     ;sino uno antes
  sub bx,8
lr_lista:
  mov izquierda,dx              ;retener en var global
  mov derecha,bx

; ax/bx : coord. iniciales izq.    (xl0/yl0)
; cx/dx : coord. iniciales dcha.   (xr0/yr0)
; si    : bandera rebase izq.
; di    : bandera rebase dcha.
; bp    : puntero al punto actual

  setnewlinel                   ;cargar variable de línea
  setnewliner

bucleIzq:
  cmp ax,xl1
  je nueva_liniaIzq             ;si fin -> nueva línea
  cmp bx,yl1
  je nueva_liniaIzq             ;sino seguir dibujando

  or si,si                      ;bandera incremente <= 0
  jg taman_band
incyl:                          ;este lugar tiene un patch !
  inc bx                        ;y seguir
  add si,dxl                    ;continuar IncFlag

  txt_incl                      ;seguir coord. 3D

  cmp bx,yl1                    ;¿meta alcanzada?
  je nueva_liniaIzq             ;nueva línea
  jmp izquierda_erhöht          ;a la izq. se aumenta y -> ahora
dcha.
taman_band:
  sub si,dyl                    ;decrementar Inkflag
incxl:                          ;este lugar tiene un patch !
  inc ax                        ;seguir x
  jmp bucleIzq
```

```
lista_:
  jmp fertig

nueva_liniaIzq:
  mov bx,izquierda          ;preparar incremento
  cmp bx,derecha
  je lista_                 ;igual, listo
  add bx,8                  ;izq. seguir
  mov ax,polyn              ;izq  al final de la lista?
  shl ax,3
  sub ax,8                  ;fin determinado
  cmp bx,ax                 ;comparación
  jb izquierda_setzen
  xor bx,bx                 ;si es SI, poner a 0
izquierda_setzen:
  mov izquierda,bx
  setnewlinel               ;cargar variables de nuevo
  jmp bucleIzq
fertig_:
  jmp fertig
izquierda_erhöht:

bucle_drcha:
  cmp cx,xrl
  je neue_linier            ;si fin alcanzado -> nueva línea
  cmp dx,yrl
  je neue_linier            ;sino seguir dibujando
  or di,di                  ;bandera incremento <= 0
  jg flagrgross
incyr:                      ;este lugar tiene un patch !
  inc dx                    ;seguir y
  add di,dxr                ;continuar IncFlag

  txt_incr

  cmp dx,yrl                ;meta alcanzada?
  je neue_linier            ;nueva línea
  jmp derecha_erhöht        ;dcha. incrementada y, dibujar línea
flagrgross:
  sub di,dyr                ;decrementar Inkflag
incxr:
  inc cx                    ;este lugar tiene un patch !
  jmp bucle_drcha

neue_linier:
  mov dx,derecha            ;preparar decremento
  cmp dx,izquierda
  je fertig_                ;si igual, listo
```

```
  sub dx,8                    ;si antes a 0->colocar al final
  jns derecha_setzen
  mov dx,polyn
  sub dx,2
  shl dx,3                    ;posicionado al final
derecha_setzen:
  mov derecha,dx

  setnewliner                 ;cargar variables de nuevo
  jmp bucle_drcha

derecha_erhöht:
  push ax
  push cx
  cmp cx,ax                   ;orden correcto?
  jae direct_ok               ;si ok, sino:
  cmp w fl_dorso,0            ;¿suprimir caras ocultas?
  je dibujar                  ;no, entonces dibujar
  pop cx
  pop ax
  jmp fertig                  ;polígono no se dibuja
dibujar:
  xchg ax,cx                  ;coordenadas en orden correcto
direct_ok:

  cmp texture,1               ;¿emplear texturas?
  jne relleno_normal          ;no, rellenar normal
  call hline_texture          ;dibujar línea de textura horizontal
  pop cx
  pop ax
  jmp bucleIzq                ;y seguir
relleno_normal:
  call hline                  ;dibujar línea horizontal
  pop cx
  pop ax
  jmp bucleIzq                ;y seguir
fertig:
  popa
  pop bp
  ret
fillpol endp
code ends
end
```

Primero se obtiene el punto más alto en el bucle *npoint* (el que tiene la coordenada *y* menor). *BX* contiene el mínimo actual y *SI* el número de este punto. Después se cargan las variables *izquierda* y *derecha*, que muestran qué línea se está procesando por la derecha o la izquierda. Si *izquierda* por ejemplo apunta al punto *1*, se está dibujando una línea desde el punto *1* al punto 2 en el borde izquierdo del polígo-

no. En el borde derecho la variable está definida al revés: un número de punto igual a *3* en la variable significa que se está dibujando un línea desde el punto *4* al punto *3*. Ahora se coloca *izquierda* en el punto alto calculado, y *derecha* en la tabla de coordenadas, sobre el punto anterior o al final de la definición, si *izquierda* apunta al punto *0*. Ahora se llaman las dos macros *setnewlinel* y *setnewliner*, que cargan las variables necesarias para la línea derecha e izquierda.

Para ello, de forma similar al algoritmo Bresenham, se calculan primero *delta x* y *delta y*, formando su suma en caso de signo negativo y modificando la parte correspondiente del código, para que se dibuje en dirección inversa. La automodificación no tiene nada que ver con la programación estructurada, y por ello es despreciada por muchos informáticos, pero aquí representa el modo más efectivo y rápido de controlar el bucle, evitando variables.

Incflag ha de indicar durante el transcurso del bucle de representación cuándo se ha de incrementar la coordenada *y*. Se incrementa con *deltax* y en el caso de una pendiente mayor de 1 (*deltax* < *deltay*) se invierte el signo. Tanto *Incflag* como la coordenada actual se guardan en registros para obtener una gran velocidad de representación. En la parte izquierda esto son los registros *AX/BX* para las coordenadas y *SI* para *Incflag* y en la parte derecha son *CX/DX* y *DI*.

El algoritmo lineal en sí emplea una especie de pendiente que se define mediante *delta x* y *delta y*. En cada movimiento en dirección x se decrementa *Incflag* (en *SI* o *DI*) en *delta y*, y en cada movimiento en dirección *y* se incrementa en *delta x*. Mediante el cambio de signo de *Incflag* se puede decidir en qué dirección se ha de realizar el siguiente paso: bandera positiva significa un paso en dirección *x*, negativa en dirección *y*.

Como ejemplo sean *delta x = 100* y *delta y = 50*, es decir, al principio se carga *Incflag* con *100*. El primer movimiento se realiza hacia la derecha (*Incflag = 100*, es decir > 0, después se resta *delta y*, es decir *50*), el segundo también (*Incflag* ahora 0). El siguiente movimiento se realiza en dirección *y*, ya que *Incflag* <= 0. Después se incrementa *Incflag* en *delta x*, de modo que vuelva a valer *100*. De esta forma se realizan aquí dos pasos hacia la derecha, y uno hacia abajo, lo que corresponde a la pendiente deseada de 0.5.

Estos cálculos se realizan en los bucles *buclei* y *bucled* para el lado izquierdo y derecho. Primero se comprueba si ya se ha alcanzado el punto de destino. Para ello se desplazó en el procedimiento *SetNewLine* el punto de destino en *1* tanto en dirección *x* como *y*. Así que en este lugar sólo se ha de comprobar si una de las dos coordenadas es igual al punto destino ampliado. En ese caso ya se sobrepasó el punto de destino original.

La siguiente secuencia hasta la etiqueta *listo* corresponde al algoritmo descrito - con una ampliación: si se incrementa la coordenada y, se bifurca a la etiqueta *izq_aument* o *dcha_aument*. El primer caso significa una entrada en el bucle de líneas *bucled*, el segundo una llamada de la línea horizontal. Esto, en texto llano, significa que la línea izquierda se dibuja hasta que se incrementa la coordenada *y*, y después se procede del mismo modo en el lado derecho. Si ambos lado han avanzado un punto en la coordenada *y*, se puede rellenar esta línea.

En cuanto una línea llega al destino, se ha de obtener la siguiente de la lista y fijar las variables correspondientes. Esto ocurre en las etiquetas *nueva_lineai* y *nueva_linead*.

Primero se comprueba si *izquierda* y *derecha* son idénticas. En ese caso se termina de dibujar el polígono y el procedimiento se puede abandonar mediante *listo_* o *listo__*. De lo contrario se avanza el puntero en el lado izquierdo (*izquierda*) en una posición (8 bytes) y en correspondencia se retrocede *derecha* una hacia atrás. Para ello naturalmente se han de tener en cuenta los bordes, es decir, que después del último punto se ha de posicionar en el primero, y viceversa. Después de haber cargado los punteros naturalmente se han de volver a calcular las variables de línea, llamando *setnewlinel* o *setnewliner* y saltando de nuevo al bucle.

Como ya hemos dicho, se dibuja la línea después de un incremento bilateral de la coordenada *y*. Esto ocurre a partir de la etiqueta *dcha_aument*. Aquí, principalmente se llama el procedimiento *hline* (a las variaciones ya llegaremos), que dibuja una línea desde el punto *AX/BX* al punto *CX/DX*. A continuación se salta de vuelta al bucle izquierdo.

Este procedimiento guarda primero las variables necesitadas y calcula la longitud del trayecto. Si se piden superficies de vidrio, como es el caso en este programa, en este lugar se instruye al GDC que realice una operación *OR* entre el antiguo contenido del Latch y los nuevos datos.

El método más sencillo de dibujar una línea horizontal es el de colocar todos los puntos de ella en el color deseado. Pero esto no tiene mucho sentido en el modo X, ya que aquí el direccionamiento de puntos individuales es muy lento. En vez de ello se ha de intentar, siempre que se pueda, colocar cuatro puntos a la vez, o al menos tantos como son posibles en la dirección (de memoria).

A partir de la etiqueta *Solidl* se calcula ahora la dirección de destino del primer byte. De la etiqueta *sin_medio* en adelante se dibuja el primer bloque, incompleto. Este se salta directamente si el número total de puntos a dibujar es menor que 4, y se necesita si el plano inicial (obtenido mediante un AND de la coordenada x con *11b*) es distinto de *0*. Para poder dibujar este bloque, se han de enmascarar primero

en *BH* el número de píxeles a activar. En el caso de un gran número de píxeles la máscara se pone en *0f* para que se enmascaren todos los píxeles de este bloque, de lo contrario sólo se activa el número necesitado de bits. Esta máscara se desplaza a la posición correcta en función del plano inicial correspondiente y, después de un decremento de ZZ se envía al Timing Sequencer.

Ahora se realiza un acceso de lectura en el offset deseado, no para obtener informaciones sobre el contenido de la memoria de la pantalla para la CPU, sino para cargar los Latches de modo que el siguiente acceso de escritura pueda operar el nuevo color con los datos correctos. Si en este momento se han de colocar menos de cuatro píxeles, se bifurca a la etiqueta *Final*. De lo contrario se han de activar todos los planos y colocar el número correspondiente de bloques al color adecuado. Esta determinación se realiza en el caso de cuerpos opacos mediante una instrucción *rep stosb*. En el caso de los cuerpos de vidrio el tema es algo más complicado. Aquí se ha de realizar un acceso de lectura para cada byte, de modo que se emplea un bucle.

Al final se colocan todos los píxeles del último bloque en la coordenada de destino. Para ello se forma una máscara correspondiente al plano de destino y se escribe el color. En la etiqueta *hline_listo* se vuelve a fijar el modo de escritura del GDC en *Sustituir* y se abandona el procedimiento.

8.6 Ocultar líneas - Hidden Line

Los cuerpos de vidrio seguramente tienen su atractivo, pero para una representación de cuerpos reales son poco adecuados. La mayoría de cuerpos son opacos, de modo que se ha de conseguir que las representaciones por ordenador de los mismos no muestren las caras ocultas.

El problema de la supresión de caras es uno de los temas más complejos de la representación tridimensional. No sólo se trata de dibujar algunas superficies y otras no. Las superficies se superponen parcialmente, de modo que se han de dibujar ambas, pero en el orden correcto. Aquí presentamos ambos métodos - ordenación y supresión.

Para suprimir caras ocultas hay un gran número de teorías que habitualmente forman un ángulo entre la cara y la normal del ojo (recta desde el ojo hasta la superficie correspondiente). Por medio de este ángulo se puede averiguar si el usuario ve el anverso o el reverso de la superficie. Este último caso significa que la cara ha de ser suprimida.

El método que aquí presentamos parte de un principio parecido, pero realiza una enorme simplificación: se supone que todas las caras están definidas en contra de

las agujas del reloj (matemáticamente positivas). De esta forma, una cara siempre se dibuja "hacia la izquierda" independientemente de su posición en el espacio. Si se ha girado tanto que el observador ve su dorso, aparece al revés, y se dibuja "hacia la derecha". Ahora simplemente se comprueba, al dibujar las líneas horizontales mediante *hline*, si su punto final se encuentra a la derecha del punto inicial. Si no es así, se ve el dorso, que ha de ser suprimido. En el programa se realiza esta comprobación en el procedimiento *FillPol* del módulo POLY.ASM después de la etiqueta *dcha_aument*:

```
dcha_aument:
  push ax
  push cx
  cmp cx,ax                    ;¿orden correcto?
  jae direct_ok                ;ok, sino
  cmp w fl_dorso,0             ;¿suprimir dorsos?
  je dibujar                   ;no, dibujarlos
  pop cx
  pop ax
  jmp listo                    ;polígono no se dibuja
dibujar:
  xchg ax,cx                   ;coordenadas en secuencia correcta
direct_ok:
```

Los registros *AX* y *CX* contienen las coordenadas x del punto inicial y final. En el caso de una superficie definida matemáticamente como positiva, *CX* ha de ser mayor que *AX*. Si es así, se dibuja directamente (etiqueta *direct_ok*) de lo contrario se finaliza el relleno (si *fl_oculto* es *TRUE*, sino se ignora la posición y después de intercambiar las coordenadas se dibuja igualmente.

Este método es completamente suficiente para cuerpos convexos, es decir, cuerpos sin "huecos", como por ejemplo dados. ¿Pero qué ocurre en el caso de cuerpos cóncavos, como por ejemplo un objeto en forma de U? Las superficies invisibles se ordenan, pero el orden no es correcto, de modo que aún se ven algunas superficies que deberían estar parcialmente ocultas por otras. Si ahora se ordenan las superficies de forma que primero se dibuje la de mayor coordenada z (la que está más atrás) y se va dibujando "hacia adelante", las nuevas superficies delanteras tapan partes del mundo ya dibujado. Como este método también se emplea en la pintura, se habla del "Painter's Algorithm".

Ordenar superficies, bien. ¿Pero cómo? Los vértices de una superficie suelen tener coordenadas z completamente distintas. Y por ello algoritmos complejos y lentos intentan compensar esto y encontrar las relaciones entre los vértices que permitan una asociación unívoca. Sin embargo es mucho más rápida la ordenación según las profundidades medias. Para ello se forma un valor promedio de las informaciones de profundidad de cada superficie, empleándose como criterio de ordena-

ción. Este método es muy inexacto, especialmente en superficies de gran extensión, pero en combinación con la supresión de caras ocultas consigue unos resultados bastante aceptables y, sobre todo, rápidos. El ya mencionado array *medio* acoge estos valores medios y guarda los punteros de superficie correspondientes. A ello se le suman los valores z acabados de la macro *z2cx* en la posición del array. El valor medio se forma en la etiqueta *polyok*, dividiendo esta suma entre la cantidad de vértices.

```
polyok:
  mov bx,middleptr              ;obtener promedio
  mov ax,middle[bx]             ;obtener suma
  mov cx,polyn
  dec cx
  cwd
  div cx                        ;dividir por número de vértices
  mov middle[bx],ax              ;rescribir
```

Si la variable *fl_sort* esta en *TRUE*, se llama el procedimiento *Quicksort* después de calcular todas las superficies. Este ordena el array *medio* según el conocido algoritmo del Quicksort. La información de profundidad (en la mitad inferior del *DoubleWord*) se emplea como criterio de ordenación. La identificación de superficie naturalmente también se ordena.

```
public quicksort
quicksort proc pascal unten,oben:word
;ordena array de medios según algoritmo QuickSort

local clave:word
local links:word
  push bx
  mov bx,unten                  ;encontrar centro
  add bx,oben
  shr bx,1
  and bx,not 3                  ;posicionar en bloques de 4
  mov dx,Medio[bx]              ;obtener llave
  mov clave,dx
  mov  ax,unten                 ;inicializar dcha. e izq. con valores
base
  mov si,ax
  mov links,ax
  mov ax,oben
  mov di,ax

  mov dx,clave
izq_cerca:
  cmp Medio[si],dx              ;mayor que clave -> seguir buscando
  jbe toca_izq
  add si,4                      ;posicionar al siguiente
```

```
    jmp izq_cerca                ;y comprobarlo
  toca_izq:
    cmp Medio[di],dx             ;menor que clave -> seguir buscando
    jae toca_dcha
    sub di,4                     ;posicionar al siguiente
    jmp toca_izq                 ;y comprobarlo
  toca_dcha:
    cmp si,di                    ;izq. <= dcha.?
    jg fin_clave                 ;no -> zona parcial ordenada
    mov eax,dword ptr Medio[si]  ;cambiar valores medios y posiciones
    xchg eax,dword ptr Medio[di]
    mov dword ptr Medio[si],eax

    add si,4                     ;mover puntero
    sub di,4
  fin_clave:
    cmp si,di                    ;izq. > dcha., seguir
    jle izq_cerca
    mov links,si                 ;guardar izq., por recursión
    cmp unten,di                 ;abajo < dcha. -> ordenar parte izq.
    jge rechts_listo
    call quicksort pascal,unten,di;ordenar mitades recursivamente
  rechts_listo:
    mov si,links                 ;arriba>izq. -> ordenar parte dcha.
    cmp oben,si
    jle links_listo
    call quicksort pascal,si,oben ;ordenar mitades recursivamente
  links_listo:
    pop bx
    ret
  quicksort endp
```

El algoritmo Quicksort divide el campo en dos mitades e intercambia elementos, hasta que en la mitad izquierda sólo queden valores mayores que el elemento medio y en la derecha sólo menores. Después se ordenan estos subcampos de la misma forma recursiva. El funcionamiento exacto se encuentra en cualquier libro de algorítmica. Al rellenar la superficie se prescinde la de conmutación del GDC, de modo que los datos llegan directamente a la memoria de vídeo y crean una cara opaca. Aquí también se puede emplear el bucle rápido *rep stosb*, ya que no es necesario cargar los Latches antes de cada acceso de escritura.

Al contrario que en 3D_GLAS.PAS, 3D_SOLID.PAS prescinde de la generación de una paleta (por ello se definen otros colores para las superficies) y ocupa las variables globales de otra forma: *Vidrio* está en *FALSE*, ya que no se desean superficies de vidrio. Además se han colocado en *TRUE* las dos variables de tratamiento de caras ocultas, *fl_oculto* y *fl_sort*. Por lo demás los dos programas son idénticos.

8.7 Proyectar sombras - sombreado con luces

Hasta ahora ya disponemos de cuerpos sólidos, que se pueden girar en el espacio de cualquier forma. Pero no hemos tenido en cuenta un aspecto importante: la iluminación. Incluyendo una fuente de luz se pueden representar los mundos tridimensionales de forma mucho más impresionante que con las rutinas actuales. Naturalmente aún no es posible un raytracing en tiempo real en los PC de la actualidad, este tipo de imágenes necesitan varios minutos para su cálculo. Así que se necesita un método más rápido que no siga cada rayo luminoso, pero que con determinadas simplificaciones ya genere bonitos efectos. Si se parte de una fuente de luz situada en el infinito, todos los rayos luminosos llegan al objeto de forma paralela. Por ello se puede calcular con un sólo vector luminoso, en vez de tener que calcular uno por cada punto de la superficie. Con esta iluminación homogénea es suficiente calcular un brillo para cada superficie que se dibujará después. ¿Pero cómo se calcula el brillo de esta superficie? Para ello se emplea un sencillo modelo: cuanto más inclinado sea el rayo luminoso con respecto a la superficie, tanto más oscura es esta. En el caso de una iluminación perpendicular, el brillo es máximo. Esto se debe a que una cantidad de energía idéntica es repartida sobre una superficie mayor.

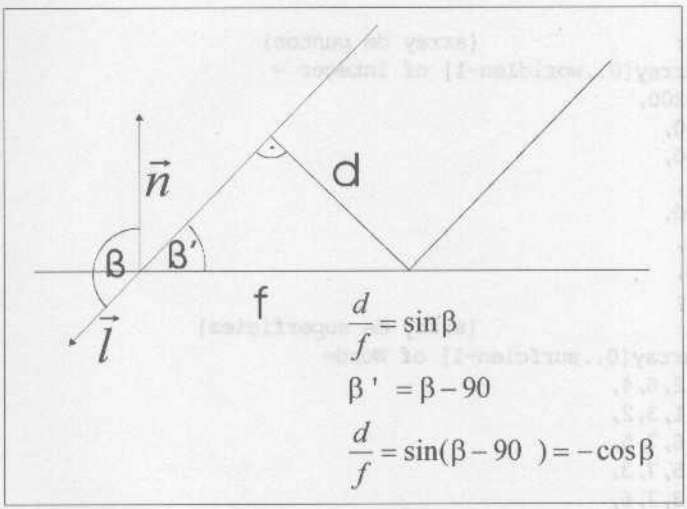

Figura 19: Relación de la intensidad de iluminación con respecto a la superficie

La relación *d/f* es proporcional al brillo de la superficie. Como muestra la fórmula, esta relación es igual al coseno negativo del ángulo entre el vector de luz y el vector normal de la superficie. El vector normal es un vector que es perpendicular a la superficie. Se puede obtener con facilidad del producto vectorial de dos vectores que se encuentran en el plano.

El hecho de que se necesite el coseno del ángulo y no el ángulo en sí, simplifica mucho los cálculos, ya que el resultado de un cálculo angular (mediante el producto escalar) es el coseno de ángulo. El modo de proceder para determinar el brillo es el siguiente:

- Encontrar dos vectores que estén en el plano. Lo más sencillo son dos vectores de los bordes (del primer al segundo punto y al último)
- Formar el vector normal (producto vectorial de los vectores de superficie)
- Formar el ángulo entre el vector de luz (constante) y el vector normal mediante el producto escalar
- Sumar el resultado al color

El resultado del cálculo del ángulo es negativo en el caso dibujado. Si la superficie está apartada de la luz (ß < 90 grados), el resultado es positivo y sólo se puede emplear el color base de la superficie, ya que se encuentra a la sombra y por ello sólo recibe luz difusa. El programa 3D_LIGHT.PAS demuestra las capacidades de la rutina de sombreado.

```
Uses Crt,ModeXLib,Gif,var_3d;

Const
    worldlen=8*3;                {array de puntos}
    Worldconst:Array[0..worldlen-1] of Integer =
    (-200,-200,-200,
    -200,-200,200,
    -200,200,-200,
    -200,200,200,
    200,-200,-200,
    200,-200,200,
    200,200,-200,
    200,200,200);
    surfclen=38;                 {array de superficies}
    surfcconst:Array[0..surfclen-1] of Word=
    ($fee0,4, 0,2,6,4,
     $fec0,4, 0,1,3,2,
     $fec0,4, 4,6,7,5,
     $fee0,4, 1,5,7,3,
     $fec0,4, 2,3,7,6,
     $fec0,4, 0,4,5,1,0,0);
    { $fe = emplear fuente de luz, color base en Low-Byte}

  Var
    i,j:Word;

  Procedure Pal_sombra;          {preparar paleta para sombreado}
  Begin
    For j:=192 to 223 do Begin   {preparar colores 192-223 y 224-255}
```

```
      i:=trunc((j/32)*43);          {obtener brillo}
      Fillchar(Palette[j*3],3,i+20); {colores 192-223 a grises}

      Palette[(j+32)*3]:=i+20;      {colores 224-255 a rojos}
      Palette[(j+32)*3+1]:=0;
      Palette[(j+32)*3+2]:=0;
    End;
    Setpal;                         {fijar esta paleta}
  End;

  procedure drawworld;external;  {dibuja el mundo en la pág. actual de
  pant.}
  {$l 3dasm.obj}
  {$l poly.obj}
  {$l bres.obj}
  {$l wurzel.obj}
  Begin
    vz:=1000;                       {cuerpo se encuentra en 1000 uds. de
  prof.}
    vpage:=0;                       {comenzar con pág. 0}
    LoadGif('logor.gif');           {cargar imagen de fondo}
    init_modex;                     {activar modo X}
    Pal_sombra;                     {calcular paleta de sombreado}
    rotx:=0;                        {valores inic. para rotación}
    roty:=0;
    rotz:=0;
    Rellenar:=true;                 {activar relleno de sup.}
    fl_sort:=true;
    fl_dorso:=true;                 {activar supresión de caras ocultas}
    Vidrio:=false;                  {apagar superficies de vidrio}
    p13_2_modex(16000*2,16000);     {fondo en pág. VGA 2}
    repeat
      CopyScreen(vpage,16000*2);    {imagen de fondo en pág. actual}
      DrawWorld;                    {dibujar mundo}
      switch;                       {conmutar a la imagen acabada}
      WaitRetrace;                  {esperar siguiente retrazado}
      Inc(rotx);                    {seguir rotando... }
      If rotx=120 Then rotx:=0;
      Inc(rotz);
      If rotz=120 Then rotz:=0;
      inc(roty);
      if roty=120 Then roty:=0;
    Until KeyPressed;               { ... hasta tecla}
    TextMode(3);
  End.
```

Antes de poder ajustar el brillo, primero se ha de preparar la paleta. Para ello se rellenan los colores *192* hasta *223* y *224* hasta *255* con dos degradados de color. El primero contiene colores desde gris hasta blanco, el segundo desde rojo oscuro hasta rojo claro. Ahora se puede sumar simplemente el coseno al color base duran-

te el relleno para obtener el brillo correcto. El color base se guarda en el byte bajo de la información de color (aquí son *0c0h* y *0e0h*), mientras que el byte alto se llena con el valor de control. Mediante *0feh* se activa el sombreado de esta superficie. Por ello es posible sin más, mezclar superficies sombreadas (byte de comando *0feh*) con superficies de valor fijo (byte de control < *0feh*) o bien añadir texturas posteriormente (byte de control *0ffh*).

Otra modificación con respecto el programa 3D_SOLID estriba en la imagen de fondo. Para comenzar, se carga una imagen que fue calculada en un raytracer y que posee un vector de iluminación similar al de los objetos en rotación, lo que incrementa la impresión de la fuente de luz. En vez de borrar la memoria de la pantalla antes de cada construcción de pantalla, simplemente se copia la imagen de fondo en la página correspondiente. En el procedimiento *Drawworld* (en *3DASM.ASM*) se descifra directamente al principio del byte de control, asignando las variables *Lightsrc* y *Texture* en correspondencia. En este caso sólo la variable *Lightsrc*.

Ahora también se emplean los arrays *Points3D* y *Poly3D*. El primero guarda las coordenadas tridimensionales, que se calcularon en la macro *zrot*. Estas se transfieren al array *Poly3D* en el bucle *npoint*, convirtiéndolas finalmente en un trazo cerrado, copiando el último vértice sobre el primero. Este proceso se corresponde completamente al que se realiza para la coordenadas bidimensionales. Si ahora se rellena el área, pero primero se llama la macro *getdelta*, que determina los dos vectores de superficie:

```
getdelta macro                  ;calcula ambos vectores de sup.
  mov ax,poly3d[0]              ;x: vértice origen
  mov delta2[0],ax              ;guardar en delta2
  sub ax,poly3d[8]              ;crear diferencia con el primer punto
  mov delta1[0],ax              ;y delta1 listo

  mov ax,poly3d[2]              ;y: vértice origen
  mov delta2[2],ax              ;guardar en delta2
  sub ax,poly3d[10d]            ;crear diferencia con el primer punto
  mov delta1[2],ax              ;y delta1 listo

  mov ax,poly3d[4]              ;z: vértice origen
  mov delta2[4],ax              ;guardar en delta2
  sub ax,poly3d[12d]            ;crear diferencia con el primer punto
  mov delta1[4],ax              ;y delta1 listo

  mov bp,polyn                  ;seleccionar último punto
  dec bp
  shl bp,3                      ;a 8 bytes
  mov ax,poly3d[bp]             ;obtener x
  sub delta2[0],ax              ;formar resta
```

```
  mov ax,poly3d[bp+2]          ;obtener y
  sub delta2[2],ax             ;formar resta
  mov ax,poly3d[bp+4]          ;obtener z
  sub delta2[4],ax             ;formar resta
endm
```

Delta1 se carga con la diferencia entre el primer y el segundo punto del polígono y *delta2* con la diferencia entre el primer y el último punto. Mientras no se definan superficies sin sentido estos vectores siempre son linealmente independientes (no paralelos) y con ello adecuados. De lo contrario, el programa se interrumpirá con un *Division by Zero*. Si en este caso la variable global *lightsrc* es *TRUE*, se llaman las dos macros *get_normal* y *light*. La primera determina el vector normal de la superficie en base a los vectores de superficie *delta1* y *delta2*. La segunda macro determina el brillo de la superficie en función al ángulo.

```
get_normal macro               ;calcula el vector perpendicular de
una sup.
  mov ax,delta1[2]             ;a2*b3
  imul delta2[4]
  shrd ax,dx,4
  mov n[0],ax
  mov ax,delta1[4]             ;a3*b2
  imul delta2[2]
  shrd ax,dx,4
  sub n[0],ax
  mov ax,delta1[4]             ;a3*b1
  imul delta2[0]
  shrd ax,dx,4
  mov n[2],ax
  mov ax,delta1[0]             ;a1*b3
  imul delta2[4]
  shrd ax,dx,4
  sub n[2],ax
  mov ax,delta1[0]             ;a1*b2
  imul delta2[2]
  shrd ax,dx,4
  mov n[4],ax
  mov ax,delta1[2]
  imul delta2[0]
  shrd ax,dx,4
  sub n[4],ax                  ;producto cruzado (=vector perp.) listo

  mov ax,n[0]                  ;x1 ^ 2
  imul ax
  mov bx,ax
  mov cx,dx
  mov ax,n[2]                  ;+x2 ^ 2
  imul ax
```

```
  add bx,ax
  adc cx,dx
  mov ax,n[4]                ;+x3 ^ 2
  imul ax
  add ax,bx
  adc dx,cx                  ;Suma en dx:ax
  push si
  call raiz                  ;raiz en ax
  pop si
  mov n_cant,ax              ;valor del vector perp. listo
endm
```

La primera parte de esta macro calcula el vector normal en sí. Para ello se forma cada componente individual del vector de la diferencia de dos productos, guardándose en el Array *n*. La segunda parte calcula ahora el valor del vector normal, elevando al cuadrado todos los componentes y sumándolos a continuación. De esta suma se extrae la raíz por aproximación y se guarda el resultado en *n_betr*. El último paso lo realiza la macro *light*.

```
light macro                  ;determina brillo de una sup.
  mov ax,n[0]
  imul l[0]                  ;vector luz * vector perp.
  mov bx,ax                  ;formar suma en cx:bx
  mov cx,dx
  mov ax,n[2]
  imul l[2]
  add bx,ax
  adc cx,dx
  mov ax,n[4]
  imul l[4]
  add ax,bx                  ;producto escalar listo en dx:ax
  adc dx,cx
  idiv l_cant                ;dividir por l_cant

  mov bx,n_cant              ;y por n_cant
  cwd
  shld dx,ax,5               ;valores de -32 a +32
  shl ax,5d
  mov bp,startpoly           ;preparar direccionamiento del color
de sup.
  idiv bx                    ;división por el divisor
  inc ax
  or ax,ax
  js encarado                ;si cos α positivo -> apartado de la
luz
  xor ax,ax                  ;así que no hay luz
encarado:
  sub b polycol,al           ;cos<0 -> sumar a color básico
endm
```

Después de haber formado el producto escalar entre el vector normal y el vector (constante) de luz (en *l*), se divide entre el valor del vector luz (igualmente constante) y el del vector normal. De esta forma se determina el ángulo entre los dos vectores. El resultado de este cálculo normalmente estaría entre mas menos uno. Pero como trabajamos con números enteros, antes de la segunda división se multiplica el resultado intermedio con *32*, para obtener un rango de valores de -32 hasta +32. Si el resultado es positivo, *AX* se pondrá a cero y con ello se mantiene el color original. Si es negativo, se resta este valor del color fundamental. El brillo calculado se encuentra ahora en la variable *PolyCol*, que es empleada por *FillPol* como color de relleno.

8.8 Superficies impresionantes - Texturas

El último y a la vez mayor paso que sufrirán las rutinas 3D de este libro es la inclusión de texturas, que pretenden darle una estructura a las superficies que hasta ahora eran lisas y monocromas. Para ello se proyectan gráficos bitmap sobre las caras de los objetos, moviéndose sincronizadamente en cada rotación. A causa de ello, los bitmaps prácticamente se "pegan" en las caras y pueden simular una superficie determinada como madera o metal. Esta técnica encuentra actualmente mucha aplicación en los juegos de rol como Ultima Underworld donde las paredes a veces son de piedra, a veces de madera o de otros materiales.

Si se piensa en un concepto concreto de programación, en la mayoría de los casos se verá primero la posibilidad más sencilla. Se puede componer un área de pequeñas áreas parciales que correspondan con un punto del bitmap. Pero esta técnica no es la más rápida, además de que sólo funciona correctamente en la minoría de los casos. Si se aumenta o gira un área, inmediatamente aparecen huecos, ya que algunos puntos se superponen, con lo que faltan justo al lado.

La única solución practicable de este problema es la de darle la vuelta al procedimiento: durante la representación de la superficie se procede del modo conocido, y con ello se activan todos los puntos. Cada uno de estos puntos se vuelve a proyectar a la superficie original, para determinar su posición dentro de esta. Con ayuda de esta posición se puede leer el color del punto del bitmap de textura.

Como es imposible deducir la posición tridimensional de un punto en función de las coordenadas bidimensionales de la pantalla, siempre se tienen en cuenta la coordenadas 3D durante el relleno, de modo que se pueden conocer la coordenadas de cada punto y por ello su posición en la superficie. Como

demostración de las rutinas de textura se emplea el programa 3D_TEXTU.PAS. Este sólo se diferencia de los anteriores en otros valores de color en la definición del mundo y en el procedimiento *Prep_Texturas*, que se llama justo al principio.

```
Uses Crt,ModeXLib,Gif,var_3d;

Const
    worldlen=8*3;                    {array de puntos}
    Worldconst:Array[0..worldlen-1] of Integer =
    (-200,-200,-200,
    -200,-200,200,
    -200,200,-200,
    -200,200,200,
    200,-200,-200,
    200,-200,200,
    200,200,-200,
    200,200,200);
    surfclen=38;                     {array de superficies}
    surfcconst:Array[0..surfclen-1] of Word=
    ($ff00,4, 0,2,6,4,
     $ff01,4, 0,1,3,2,
     $ff02,4, 4,6,7,5,
     $ff00,4, 1,5,7,3,
     $ff03,4, 2,3,7,6,
     $ff04,4, 0,4,5,1,0,0);

    { $ff = emplear texturas, nº en Low-Byte}

Var
    i,j:Word;

Procedure Prep_Texturas;
{cargar variables de las texturas}
Begin
  LoadGif('Textur');                 {cargar imagen de textura}
  GetMem(Txt_Pic,64000);             {obtener memoria para ella}
  Move(VScreen^,Txt_Pic^,64000);{y copiar allí}
  For i:=0 to Txt_numero-1 do Begin
    Txt_Datos[i]:=Txt_Pic;           {cargar puntero a datos}
    Txt_Offs[i]:=i*64;               {determinar offset}
  End;
End;

procedure drawworld;external;  {dibuja el mundo en la pág. de pant.
actual}
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l wurzel.obj}
```

```
Begin
  vz:=1000;                        {cuerpo se encuentra en 1000 uds. de
prof.}
  vpage:=0;                        {comenzar con pág. 0}
  init_modex;                      {activar modo X}
  Prep_Texturas;
  LoadGif('logo.gif');             {cargar imagen de fondo}
  rotx:=0;                         {valores inic. para rotación}
  roty:=0;
  rotz:=0;
  Rellenar:=true;                  {activar relleno de sup.}
  fl_sort:=true;                   {activar ordenación de sup.}
  fl_dorso:=true;                  {activar supresión de caras ocultas}
  Vidrio:=false;                   {apagar superficies de vidrio}
  p13_2_modex(16000*2,16000);      {fondo en página VGA 2}
  repeat
 CopyScreen(vpage,16000*2);        {imagen de fondo en página actual}
  DrawWorld;                       {dibujar mundo}
  switch;                          {conmutar a imagen terminada}
  WaitRetrace;                     {esperar siguiente retrazado}
  Inc(rotx);                       {seguir rotando... }
  If rotx=120 Then rotx:=0;
  Inc(rotz);
  If rotz=120 Then rotz:=0;
  inc(roty);
  if roty=120 Then roty:=0;
 Until KeyPressed;                 { ... hasta tecla}
 TextMode(3);
End.
```

Como byte de control se emplea para todas las superficies el valor *0ffh*, que para cada superficie correspondiente activa la textura especificada en el byte bajo. El procedimiento *Prep_Texturas* carga primero la imagen GIF con las texturas necesarias y la mueve a otra zona de memoria (*TxtPic^*). En este lugar entran en juego los arrays *Txt_Datos* y *Txt_offs*. *Txt_Datos* indica un puntero para cada textura con la posición de la pantalla de textura, de esta manera es posible cargar diferentes pantallas independientes con texturas. *Txt_offs* contiene el offset de la textura correspondiente dentro de esa pantalla. En este caso, *Txt_Datos* contiene un puntero a *Txt_Pic^* para cada una de las texturas y *Txt_offs* un múltiplo de *64*, es decir cinco texturas adyacentes de 64 píxeles de anchura.

El tamaño de las texturas se determina mediante el array de constantes *Txt_Tam*. Aquí se determina para cada textura el tamaño *y* mediante el byte alto y el tamaño *x* mediante el byte bajo. Sin embargo se ha de tener en cuenta una norma especial a causa del cálculo de estos tamaños: las texturas tienen respectivamente unas medidas de *256 shr (n-8)* píxeles. El valor *10 (0ah)* significa por ejemplo un tamaño de *256 shr 2 = 64* píxeles (en dirección *x* o *y*).

Algunas partes de programa ya comentadas, como la formación de los vectores de superficie *Delta1* y *Delta2* vuelven a encontrar aplicación en las texturas. La mayor parte de los algoritmos necesarios se encuentra en forma de una macro en el archivo include TEXTURE.INC, que funciona como extensión de POLY.ASM.

En el caso de superficie de textura, la variable global *Texture* está a *TRUE*, por ello la determinante principal se forma directamente al principio del procedimiento *FillPol* en la macro *Txt_Detprinc*. Con su ayuda se podrá determinar después la coordenada relativa del punto en la superficie. Para ello se ha de saber que cada punto de una superficie es una solución del siguiente sistema de ecuaciones:

```
x1= lambda1 * a1 + lambda2 * b1
x2= lambda1 * a2 + lambda2 * b2
x3= lambda1 * a3 + lambda2 * b3,
```

X1 - x3 son las coordenadas del punto, *a1 - a3* las componentes del primer vector de superficies (*Delta1*) y *b1 - b3* los del segundo (*Delta2*). *Lambda1* y 2 indican las coordenadas afines relativas a los dos vectores de superficie y se pueden emplear para el acceso directo a la textura. Para poder calcular *lambda1* y *lambda2*, simplemente se necesitan dos de las ecuaciones, la tercera se cumple en cualquier caso, ya que el punto se encuentra en el plano.

Si por ejemplo se toman las dos primeras ecuaciones, se puede encontrar la solución de forma muy sencilla mediante determinantes: la determinante principal es $D = a1 * b2 - a2 * b1$, la primera codeterminante es $D1 = x1 * b2 - x2 * b1$ y la segunda $D2 = a1 * x2 - a2 * x1$. Las dos incógnitas resultan de $lambda1 = D1/D$ y $lambda2 = D2/D$. La determinante principal es igual para toda el área, de modo que se puede calcular directamente:

```
txt_Hauptdet macro                  ;calcular determinante principal
    xor si,si                       ;1§ intento: línea 0 y 1
    mov di,2

naechste:
    mov ax,w delta1[si]             ;calcular determinante principal
    imul w delta2[di]
    mov bx,ax                       ;guardar resultado intermedio
    mov cx,dx
    mov ax,w delta2[si]
    imul w delta1[di]
    sub bx,ax                       ;guardar diferencia
    sbb cx,dx
    mov w D,bx
    mov w D+2,cx
    or bx,cx                        ;determinante principal = 0?
    jne D_fertig
    add si,2                        ;entonces nuevos componentes
```

```
  add di,2
  cmp di,4                    ;en las l¡neas existentes?
  jbe naechste
  xor di,di                   ;no, comenzar por arriba
  jmp naechste
D_fertig:
  movsx eax,delta1[si]        ;guardar valores de columna empleados
  mov spalte1[0],eax
  movsx eax,delta1[di]
  mov spalte1[4],eax
  movsx eax,delta2[si]
  mov spalte2[0],eax
  movsx eax,delta2[di]
  mov spalte2[4],eax
  shl si,1                    ;guardar columnas empleadas
  shl di,1
  mov obere_Reihe,si
  mov untere_Reihe,di
endm
```

Sin embargo aparece un problema: bajo determinadas circunstancias la elección de las dos primeras ecuaciones puede ser poco favorable, lo que se demuestra por que la determinante principal da *0*. En este caso se posiciona simplemente el vector de superficie en la siguiente línea (primero 0/2, después 2/4, después 4/0) y se hace un nuevo intento. Después se guarda la determinante principal en los arrays *Columna1* y *Columna2* y los números de línea empleados en *fila_superior* y *fila_inferior*. Estos datos se necesitarán posteriormente.

Como ya hemos dicho, el algoritmo de relleno siempre ha de vigilar en qué coordenadas se encuentra el punto actual. Para ello se calculan líneas tridimensionales paralelas a los vértices del polígono que lo describan. En las macros *SetNewLineL* y *SetNewLineR* se ha de incluir la inicialización de la línea tridimensional correspondiente. Esto se realiza mediante la macros *Txt_MakeVarL* y *Txt_MakeVarR*.

```
 txt_makevarl macro           ;carga de nuevo la var. 3D parte
izq.
.386
  movsx ebx,dyl               ;n$ de pasos
  inc ebx
  push ecx
  push edx

  movsx eax,poly3d[bp]        ;obtener 3D-x
  shl eax,8                   ;bit inferiores son "decimales"
  mov xl_3d,eax               ;y escribir
  movsx ecx,poly3d[bp+8]      ;formar diferencia
```

```
    shl ecx,8
    sub eax,ecx
    neg eax
    cdq
    idiv ebx    ;determinar amplitud de paso
    mov inc_xl,eax

    movsx eax,poly3d[bp+2]  ;obtener 3D-y
    shl eax,8
    mov yl_3d,eax
    movsx ecx,poly3d[bp+10d]      ;formar diferencia
    shl ecx,8
    sub eax,ecx
    neg eax
    cdq
    idiv ebx                      ;determinar amplitud de paso
    mov inc_yl,eax

    movsx eax,poly3d[bp+4]         ;obtener 3D-z
    shl eax,8
    mov zl_3d,eax
    movsx ecx,poly3d[bp+12d]       ;formar diferencia
    shl ecx,8
    sub eax,ecx
    neg eax
    cdq
    idiv ebx                      ;determinar amplitud de paso
    mov inc_zl,eax
    pop edx
    pop ecx
endm
```

En el array *Poly3D* se encuentran las coordenadas tridimensionales de los vértices de la superficie actualmente en proceso. El registro *BP* sigue apuntando a la posición de la coordenada actual en el array *Poly2D* (sigue encontrándose en *SetNewLineL* o *SetNewLineR*). Ya que estos arrays están colocados de forma completamente síncrona, se pueden direccionar las coordenadas 3D de los vértices mediante *BP*. La macro carga las variables *xl_3D, yl_3D*, etc. con los valores iniciales del primer vértice y forma la diferencia al segundo. Esto es la longitud de la línea, en dirección *x, y* y *z*.

El principio de estas líneas se basa en que durante el dibujo se avanza un paso en la línea con cada modificación de la coordenada *y*. Así que se han de dividir las diferencias entre el número de pasos (es decir la "altura" de la línea bidimensional en *dyl* o *dyr*). Todos los valores se desplazan 8 bits hacia la izquierda, para obtener una precisión suficiente. Si la línea tridimensional mide por ejemplo 7 unidades (por ejemplo en dirección *z*) y el número de pasos es de cuatro, se ha de añadir cada vez un valor de 1.75, lo que sólo es posible si se reserva el byte bajo

para la parte decimal (en este caso *inc_zl = 01c0h*). Ahora naturalmente se han de incrementar las variables hasta la siguiente línea de pantalla en cada paso. Para ello se incluye, después de las etiquetas *incyl* o *incyr* la macro *Txt_IncL* o *Txt_IncR*:

```
txt_incl macro                  ;seguir por la izq.
  push eax
  mov eax,inc_xl                ;sigue coord. 3D-x
  add xl_3d,eax
  mov eax,inc_yl                ;sigue coord. 3D-y
  add yl_3d,eax
  mov eax,inc_zl                ;sigue coord. 3D-z
  add zl_3d,eax
  pop eax
endm
txt_incr macro                  ;seguir por la dcha.
  push eax
  mov eax,inc_xr                ;sigue coord. 3D-x
  add xr_3d,eax
  mov eax,inc_yr                ;sigue coord. 3D-y
  add yr_3d,eax
  mov eax,inc_zr                ;sigue coord. 3D-z
  add zr_3d,eax
  pop eax
endm
```

Estas macros no hacen otra cosa que lo que su nombre ya indica: incrementan la variable contadora correspondiente. *inc_xl* se suma a *xl_3D*, *inc_yl* a *yl_3D* etc. Con ello el bloque de variables *xl_3D - yl_3D - zl_3D* siempre obtiene la coordenada actual 3D del punto inicial izquierdo de la línea de relleno horizontal. Lo mismo se aplica para el lado derecho con el punto final de la línea de relleno. El procedimiento *hline_Texture*, contenido en la macro *txt_hline* para poder colocarlo en el archivo *Include*, es más interesante. Este procedimiento es llamado para las superficies texturadas en vez de *hline*. Fundamentalmente hace lo mismo: traza una línea horizontal entre las coordenadas especificadas en *AX/BX* y *CX/DX* con una diferencia: el color no es el mismo para cada punto, sino que se obtiene de la textura.

Después de guardar las coordenadas se calculan las codeterminantes para los lados izquierdo y derecho, así como *lambda1 y lambda2*. El lado izquierdo devuelve el valor inicial, ya que se comienza a dibujar en este lado. El lado derecho devuelve el valor final, que no se guarda directamente, sino que va a parar a las variables *inc_lambda1* y *inc_lambda2*. Estas variables funciones de forma similar a las líneas tridimensionales: en cada paso a la derecha se incrementa *lambda1* en *inc_lambda1* y *lambda2* en *inc_lambda2*:

```
txt_hline macro
hline_texture proc near          ;sustituye procedim. "hline" en tex-
turas
.386
  push es
  pusha
  mov x0,ax                      ;guardar coordenadas para despu,s
  mov y0,bx
  mov xl,cx
  sub cx,ax                      ;calcular n§ de puntos a dibujar
  jne zzok2
  inc cx
zzok2:
  mov zz,cx

  mov bp,obere_Reihe
  mov bx,untere_Reihe
  mov eax,xr_3d[bx]              ;determinar coord. x relativa
  movsx ecx,poly3d[2]
  shl ecx,8                      ;pasar a formato "coma fija"
  sub eax,ecx
  mov d_y,eax
  movsx ecx,w Spalte2[0]
  imul ecx                       ;multiplicar con Delta2 x•
  mov esi,eax                    ;guardar resultado

  mov eax,xr_3d[bp]              ;determinar coordenada relativa y
  movsx ecx,poly3d[0]
  shl ecx,8                      ;pasar a formato "coma fija"
  sub eax,ecx
  mov d_x,eax
  movsx ecx,w Spalte2[4]
  imul ecx                       ;multimplicar con Delta2 y
  sub eax,esi                    ;formar diferencia (D1)
  cdq                            ;preparar divisi¢n
  idiv dword ptr D               ;dividir por la determinante princi-
pal
  shl eax,8
  neg eax
  mov inc_lambda1,eax            ;guardar para resta

  mov eax,d_x                    ;obtener coordenada relativa x
  movsx ecx,w Spalte1[4]
  imul ecx                       ;mutliplicar con Deltal y
  mov esi,eax                    ;guardar resultado

  mov eax,d_y                    ;obtener coordenada realtiva y
  movsx ecx,w Spalte1[0]
  imul ecx                       ;multiplicar con Deltal x
  sub eax,esi                    ;formar diferencia (D2)
```

```
cdq                          ;preparar divisi¢n
idiv dword ptr D             ;dividir por la determinante princi
  pal
shl eax,8
neg eax
mov inc_lambda2,eax          ;guardar para resta
```

Con ayuda de las variables *fila_superior* y *fila_inferior* se leen las coordenadas necesitadas del lado derecho (según la determinante principal empleada), formándose la diferencia con el origen de la superficie (punto *0*). Mediante la multiplicación con la parte correspondiente de la determinante principal (en *Columna1* y *Columna2*) y la subsiguiente resta de ambos productos, se obtienen las codeterminantes que se dividen entre la principal. El resultado se guarda temporalmente en *inc_lambda1* y *inc_lambda2*. Después se obtienen de forma similar los valores *Lambda* izquierdos:

```
mov eax,xl_3d[bx]            ;determinar coord. x relativa
movsx ecx,poly3d[2]
shl ecx,8                    ;pasar a formato "coma fija"
sub eax,ecx
mov d_y,eax
movsx ecx,w Spalte2[0]
imul ecx                     ;multiplicar con Delta2 x
mov esi,eax                  ;guardar resultado

mov eax,xl_3d[bp]            ;determinar coord. relativa y
movsx ecx,poly3d[0]
shl ecx,8                    ;pasar a formato "coma fija"
sub eax,ecx
mov d_x,eax
movsx ecx,w Spalte2[4]
imul ecx                     ;multimplicar con Delta2 y
sub eax,esi                  ;formar diferencia (D1)
cdq                          ;preparar divisi¢n
idiv dword ptr D             ;dividir por la determinante pral
shl eax,8
neg eax
mov lambda1,eax              ;Lambda1 determinado
sub inc_lambda1,eax

mov eax,d_x                  ;obtener coordenada relativa x
movsx ecx,w Spalte1[4]
imul ecx                     ;multiplicar con Delta1 y
mov esi,eax                  ;guardar resultado

mov eax,d_y                  ;obtener coord. relativa y
movsx ecx,w Spalte1[0]
imul ecx                     ;multiplicar con Delta1 x
sub eax,esi                  ;formar diferencia (D2)
cdq                          ;preparar divisi¢n
```

```
idiv dword ptr D          ;dividir por la determinante principal
neg eax
shl eax,8
mov lambda2,eax           ;Lambda2 determinada
  sub inc_lambda2,eax
```

El resultado de estos cálculos *Lambda* se guarda en las variables *Lambda1* y *Lambda2* y las diferencias al anterior se calculan en *Inc_Lambda1* y *Inc_Lambda2*. Después se dividen ambos valores *Inc* entre la longitud de la línea horizontal, es decir, el número de pasos:

```
movsx ecx,zz              ;calcular pasos lambda
mov eax,inc_lambda1       ;obtener longitud total
cdq
idiv ecx                  ;dividir entre número de pasos
mov inc_lambda1,eax
mov eax,inc_lambda2       ;obtener longitud total
cdq
idiv ecx                  ;dividir entre número de pasos
mov inc_lambda2,eax
```

A causa de ello, es suficiente sumar en cada paso *inc_Lambda1* a *Lambda1* y *Inc_Lambda2* a *Lambda2*, para disponer de la *Lambda1* y 2 actuales para cada punto. En este lugar también se realizan los preparativos de direccionamiento de los puntos en la memoria de vídeo.

```
mov ax,80d                ;determinar offset
mov bx,y0
mul bx
mov bx,x0                 ;(x div 4) + y * 80
shr bx,2
add ax,bx
add ax,vpage
mov di,ax
mov ax,0a000h             ;cargar segmento VGA
mov es,ax

mov cx,x0                 ;enmascarar Start-Plane
and cx,3
mov ax,1
shl ax,cl                 ;activar bit correspondiente
mov b plane+1,al
shl al,4                  ;ampliar a High-Nibble
or b plane+1,al
```

Después de calcular el offset del primer punto, se determina el plano inicial y se guarda en el byte alto de la variable *Plane*. El byte bajo contiene constantemente 2, de modo que simplemente se ha de enviar la palabra completa al Timing Secuencer,

para seleccionar el plano actual. El plano además se prepara para la rotación, conteniendo tanto el nibble alto como el bajo la máscara del mismo (el plano *3* sería *88h*, después de una rotación *11h*, es decir, plano *0*).

```
mov bp,Txt_Nr                    ;obtener número de la textura actual
shl bp,1                         ;2 bytes por entrada
mov bx,Txt_Tamanyo[bp]           ;obtener indicación de tamaño
mov b cs:Tamanyo_Patch+3,bl      ;colocar en el código
mov b cs:Tamanyo_Patch+7,bh

mov ax,word ptr Txt_Offs[bp]     ;obtener offset de la textura

push ds
shl bp,1                         ;entradas de 4 bytes
lds si,dword ptr Txt_Datos[bp]   ;obtener puntero a datos
add si,ax
mov w cs:Ofs_Patch+2,si          ;colocar en el código

mov dx,3c4h                      ;Timing Sequencer
mov ebp,lambda1                  ;registro en vez de variable
mov esi,lambda2
```

Con ayuda del número de la textura actual (en *Txt_Nr*) se carga ahora el tamaño de la misma y se coloca en el código. Se sobrescriben los operandos de dos instrucciones *SAR*. Con ello se limita el rango de valores de las *Lambdas*. El offset también se lee del array correspondiente y se coloca en el código, junto con la parte de offset del puntero *txt_ datos*. El bucle para dibujar los puntos se puede mantener relativamente corto, lo que es muy importante para la velocidad:

```
lp:                              ;se recorre por cada punto
  add ebp,inc_lambda1            ;Lambda1 y 2 siguen
  add esi,inc_lambda2

  mov ax,plane                   ;obtener plane
  out dx,ax                      ;y seleccionar

  mov eax,ebp                    ;determinar offset en el gráfico de
textura
  mov ebx,esi
Groesse_Patch:
  sar eax,11d                    ;ajustar tamaño, se modifica
  sar ebx,11d
   imul eax,320d
  add ebx,eax

Ofs_Patch:
  mov al,ds:[bx+1111h]           ;coger color de textura, se modifica
  mov es:[di],al                 ;guardar color
  dec cx                         ;reducir nº de puntos
```

```
    je hlt_fertig              ;¿todos los puntos listos?
    rol b plane+1,1            ;siguiente plane
    cmp b plane+1,11h          ;rebase de plane de 3 a 0 ?
    jne lp                     ;no, seguir
    inc di                     ;si no aumentar offset
    jmp lp                     ;y seguir
  hlt_fertig:
    pop ds
    popa
    pop es
    ret
  hline_texture endp
  endm
```

El registro *EBP* contiene en el bucle *Lambda1* y *Lambda2*. Estos dos valores primero se han de aumentar en sus incrementos. Después se selecciona el plano actual y mediante *EBP* y *ESI* se determina el offset en la textura. Los dos valores *Lambda* se desplazan hacia la derecha en su factor de tamaño, para cubrir el rango de valores deseado.

En la etiqueta *Ofs_Patch* finalmente se realiza lo decisivo: aquí se obtiene el color de la textura. El offset colocado en el código se suma al calculado y de este lugar se lee la textura. El color se escribe en la memoria de vídeo y el bucle se cierre mediante una rotación de la máscara de planos. Si se realiza un rebase del plano *3* al plano *0*, se incrementa la dirección destino en *1*, para direccionar el siguiente offset.

9. Protección moderna contra copias - la clave de acceso

Un fenómeno del mundo de los ordenadores es que hay más máquinas vendidas, que programas. De esto se puede deducir una tasa de copias piratas muy elevada, ya que no todos los usuarios de ordenadores emplean sólo Shareware. Así que se puede entender perfectamente si las empresas de software se protegen mediante los más diversos tipos de protección contra una expansión ilegal de sus programas, ya que las pérdidas anuales de las empresas de software son de varios miles de millones de pesetas.

La forma más extendida de protección contra copias es la clave de acceso. Probablemente es la forma más antigua y la más fácil de realizar. Los ordenadores más nuevos le ofrecen la posibilidad de proteger su ordenador mediante una clave de acceso en ROM, que se pide al iniciar el sistema. Si ya ha trabajado con un ordenador en red, seguramente ya habrá sido confrontado con las claves, que le permiten el acceso a determinadas zonas (o no). Si es propietario de un módem, ya debe conocer las claves de entrada a las BBS. En los siguientes apartados nos ocuparemos de qué posibilidades tiene para implementar una clave de acceso, y cuál puede ser la más adecuada para usted.

9.1 Así protege sus programas - claves de acceso

Existen diferentes posibilidades para proteger su programa. Primero ha de decidir para qué quiere emplear la clave de acceso. Podríamos dividir las posibilidades de aplicación en tres grandes grupos:

1. La clave de acceso como derecho de entrada

La clave sirve como reconocimiento de que un usuario conocido maneja el programa y no una persona externa. Este tipo de claves se pueden encontrar, por ejemplo, en las BBS en sus cuentas de usuarios, en redes o en programas comerciales de los que se emplean en bancos o grandes empresas. Al fin y al cabo no a todos los empleados le importan las ventas de la empresa, almacenadas en el departamento de gestión...

2. La clave de acceso como registro

Este tipo de clave se emplea sobre todo en productos de Shareware. Después del registro se le envía una clave, con la que puede ampliar el programa de la versión Shareware a la completa. Este método se emplea poco en juegos, aunque sí en aplicaciones.

3. La clave de acceso como protección de copia

Este tipo de claves se emplea frecuentemente en juegos. Se basa en que se copia el juego, pero no el manual. Frecuentemente la clave no se pide al inicio del programa, sino al final de un nivel, o cuando se quiere cargar una partida guardada. Este último método es el que más se emplea actualmente para proteger juegos de copias indeseadas.

Otras posibilidades, como por ejemplo la necesidad de un disquete de arranque que contiene algunos sectores modificados, han pasado un poco de moda, ya que frecuentemente son consideradas como molestas por el usuario. Además de no corresponder ya a estos tiempos de los discos de 340 MBytes. En programas muy caros se puede encontrar frecuentemente un *Dongle*. El Dongle es un pequeño conector que se coloca en el puerto paralelo o serie y que contiene una clave o suma de control. Algunas versiones especiales incluso contienen rutinas de programa completas. La desventaja de este tipo de protección es que con el empleo de varios de ellos se puede formar rápidamente una "cola" detrás del PC, que puede provocar problemas de emplazamiento, sobre todo en oficinas.

Pero aquí no nos ocuparemos de las claves de hardware, sino sólo de las soluciones puras de software. Para los tres tipos de clave se aplica una máxima: cuanto mejor oculta la clave y cuanto mejor codificada, tanto más seguro es su programa. Debería decidir exactamente dónde guarda su clave. A ser posible elija un archivo externo, y no el ejecutable. Si trabaja con varios archivos, guarde la clave en uno de los más grandes. En ningún caso se debería encontrar al principio o al final del archivo, ya que ahí es donde se encuentra con más facilidad. Especialmente si trabaja con un solo archivo de datos, debería guardar la clave en uno de los campos centrales. Allí es más difícil de localizar. A ser posible no guarde directamente la cadena a comparar, sino una versión codificada de la misma. Asegúrela además por una suma de control. La mayoría de claves funcionan según el siguiente esquema:

- Construir pantalla
- Leer código del usuario

- Comparar con la clave y reaccionar en consecuencia

Este método funciona estupendamente y es muy sencillo de programar. Por ello lo encontrará en el 95 % de los casos. Si quiere proteger un programa propio con una clave, debería proceder de otra forma. Mediante una división de los diferentes pasos, el código no es tan fácil de comprender. Así que construya primero la pantalla, haga entonces lo que deba hacer: abrir archivos, inicializar variables, desviar vectores, comprobar sumas de control, etc. Sólo después de todo ello lea la clave de acceso. A causa de la alta (?) velocidad de trabajo del PC y el buffer de teclado, no perderá ninguna introducción del usuario. Después de haber leído el código, no debería realizar la comparación directamente, sino realizar alguna de las acciones anteriormente mencionadas. Sólo entonces debería cargar la clave del archivo, descifrarla y compararla.

El método presentado tiene la ventaja de que así le complica la vida a posibles Hackers. Si coloca la consulta de clave en un procedimiento, este se puede localizar rápidamente por la salida de pantalla y el salto condicionado por la comparación se puede convertir rápidamente en un salto incondicional en las manos de un Freak. Si se orienta en el método presentado, no posee un sistema absolutamente seguro, pero al menos ha complicado bastante la posibilidad de evitar el control de la clave ya que es más difícil de localizar. Según el tipo de clave que necesite, ha de incluir en su programa rutinas de codificación y decodificación, así como las rutinas necesarias para la carga y la grabación. Dediquémonos de momento al segundo tipo, donde sólo existe una clave. Es generada por el programador y se coloca en uno de los archivos. En el programa en sí se encuentra una rutina con la que se consulta y compara la clave.

Puede encontrar un programa para la generación de una clave codificada en PASSWGEN.PAS. Aquí se lee la clave deseada y a los diferentes caracteres se les suma un valor llave. A continuación se genera el complementario de los caracteres con *XOR 255*. Primero se guarda la llave (*1 Char*), después la clave (*256 Char*) y finalmente la suma de control (*1 Word*) que impide manipulaciones del archivo de claves. El programa genera el archivo PASSWORT.DAT que contiene la clave codificada y que se debería linkar en otro archivo.

```
program generar_archivo_de_clave;
{
 El programa encripta una clave introducida y la guarda en el archivo
 PASSWORT.DAT.
```

```
 (c)'94 by DATA BECKER                          Autor: Boris
Bertelsons
}

uses crt;
VAR Clave : string;
    pwf   : file;
    check : word;
    key   : char;

Function encripta(pasw : string;add : char) : string;
var li : integer;
begin;
  for li := 1 to 255 do begin;
    pasw[li] := char(255 xor (ord(pasw[li]) + ord(add)));
  end;
  Encripta := Pasw;
end;

Function Gen_SumaControl(pasw : string) : word;
VAR suma : word;
    li : integer;
begin;
  suma := 0;
  for li := 1 to ord(pasw[0]) do begin;
    suma := suma + ord(pasw[li]);
  end;
  Gen_SumaControl := suma;
end;

Function desencripta(pasw : string;add : char) : string;
var li : integer;
begin;
  for li := 1 to 255 do begin;
    pasw[li] := char((255 xor ord(pasw[li])) - ord(add));
  end;
  desencripta := Pasw;
end;

Function SumaControl_Ok(pasw : string; Key: char; suma : word) :
boolean;
Var tsuma : word;
    li : integer;
    h  : char;
begin;
  tsuma := 0;
  for li := 1 to ord(pasw[0]) do begin;
    tsuma := tsuma + ord(pasw[li]);
  end;
  if suma = tsuma then
```

```
    SumaControl_Ok := true
  else
    SumaControl_Ok := false;
end;

begin;
  clrscr;
  writeln('Introduzca la clave a encriptar');
  write('Clave: ');
  readln(Clave);
  writeln('Introduzca la llave (Cualquier carácter ASCII)');
  write('Key: ');
  readln(Key);
  writeln('escribiendo información a archivo ...');
  writeln;

  Check := Gen_SumaControl(Clave);
  Clave := encripta(Clave,Key);

  assign(pwf,'passwort.dat');
  rewrite(pwf,1);
  blockwrite(pwf,key,1);
  blockwrite(pwf,Clave,256);
  blockwrite(pwf,check,2);
  close(pwf);

  reset(pwf,1);
  blockread(pwf,key,1);
  blockread(pwf,Clave,256);
  blockread(pwf,check,2);
  close(pwf);

  writeln('Releyendo información para verificar...');
  writeln('Codificado : ',Clave);
  writeln('Llave: ',Key);

  Clave := desencripta(Clave,Key);
  writeln('Verify  : ',Clave);
  If SumaControl_Ok(Clave,Key,check) then
    writeln('Suma de control O.K.')
  else
    writeln('¡Atención! La suma de control es incorrecta!');

  repeat until keypressed; readkey;
end.
```

Para poder emplear la clave en su programa, primero ha de realizar un *SEEK* a la posición de programa en la que ha guardado la clave. Entonces cargue primero el *Key* (la llave), después la clave y finalmente la suma de control, tal y como

lo puede ver realizado en el programa anterior. Emplee el procedimiento decodifica para decodificar la clave, y compruebe mediante *Checksum_Ok* si se modificó la clave.

Si necesita el primer tipo de clave, ha de darla al usuario la posibilidad de elegir e introducir libremente la clave. El modo de proceder es análogo al anteriormente presentado. Guardará una clave vacía en el archivo, que el usuario podrá sobrescribir. En este tipo de clave es especialmente importante que no sea fácil de encontrar y que no se muestre en pantalla durante la introducción. Esto es necesario para que la clave sirve más para la protección contra cousuarios, que no contra copias piratas. En el siguiente programa podrá ver realizada la consulta de clave. La clave por defecto es *DATA BECKER*.

```
program Clave_Tipo1;

uses crt,design;

VAR clave : string;
    Control_Clave : boolean;

procedure secret_readln(var s : string);
var c : char;
    li : integer;
begin;
   repeat
   c := readkey;
   if c <> #8 then begin;
     s := s + c;
     gotoxy(24,12);
     for li := 1 to length(s) do write('*');
  end else begin;
     s := copy(s,1,length(s)-1);
     gotoxy(24,12);
     for li := 1 to length(s) do write('*');
     write(' ');
     gotoxy(wherex-1,wherey);
   end;
  until c = #13;
end;

procedure Cambiar_Clave;
begin;
  save_screen;
  Ventana(10,8,60,7,' Cambiar clave ',black,7);
  writexy(13,10,'Introduzca su nueva clave');
  writexy(13,12,'Clave: ');
  clave := '';
```

```
  secret_readln(clave);
  restore_screen;
  textcolor(7);
  textbackground(black);
end;

Function Gen_SumaControl(pasw : string) : word;
VAR suma : word;
    li : integer;
begin;
    suma := 0;
    for li := 1 to ord(pasw[0]) do begin;
    suma := suma + ord(pasw[li]);
   end;
   Gen_SumaControl := suma;
end;

Function encripta(pasw : string;add : char) : string;
var li : integer;
begin;
   for li := 1 to 255 do begin;
    pasw[li] := char(255 xor (ord(pasw[li]) + ord(add)));
  end;
  Encripta := Pasw;
end;

procedure Guardar_Clave;
var pwf : file;
    key : char;
    check : word;
begin;
   Check := Gen_SumaControl(Clave);
   Clave := encripta(Clave,Key);
   assign(pwf,'passwort.dat');
   rewrite(pwf,1);
   blockwrite(pwf,key,1);
   blockwrite(pwf,clave,256);
   blockwrite(pwf,check,2);
   close(pwf);
end;

procedure Consultar_Clave;
begin;
   save_screen;
   Ventana(10,8,60,7,' Consulta de clave ',black,7);
   writexy(13,10,'Introduzca su clave');
   writexy(13,12,'Clave: ');
   clave := '';
   secret_readln(clave);
```

```
   restore_screen;
   textcolor(7);
   textbackground(black);
end;

Function desencripta(pasw : string;add : char) : string;
var li : integer;
begin;
  for li := 1 to 255 do begin;
    pasw[li] := char((255 xor ord(pasw[li])) - ord(add));
  end;
  desencripta := Pasw;
end;

Function SumaControl_Ok(pasw : string; Key: char; suma : word) :
boolean;
Var tsuma : word;
    li : integer;
    h : char;
begin;
   tsuma := 0;
   for li := 1 to ord(pasw[0]) do begin;
    tsuma := tsuma + ord(pasw[li]);
   end;
   if suma = tsuma then
    SumaControl_Ok := true
   else
    SumaControl_Ok := false;
end;

procedure ComprobarClave;
var pwf : file;
    key : char;
    check : word;
    Debe_Clave : string;
begin;
   assign(pwf,'passwort.dat');
   reset(pwf,1);
   blockread(pwf,key,1);
   blockread(pwf,Debe_Clave,256);
   blockread(pwf,check,2);
   close(pwf);
   Debe_Clave := desencripta(Debe_Clave,Key);
    if SumaControl_Ok(Debe_Clave,Key,check) and (Debe_Clave = Clave)
   then
    Control_Clave := true
   else
   Control_Clave := false;
end;
```

```
procedure ReaccionarClave;
begin;
  save_screen;
  Ventana(10,8,40,7,'',black,7);
  If Control_Clave then begin;
     writexy(13,11,'Clave correcta - Acceso permitido');
  end else begin;
     writexy(13,11,'Clave ERRONEO - ¡No hay acceso!');
  end;
  repeat until keypressed; readkey;
  restore_screen;
  textcolor(7);
  textbackground(black);
end;
procedure menu;
var choice : byte;
begin;
  repeat
     clrscr;
     writexy(10,1,'Programa de ejemplo para clave tipo 1 (C) ''94 by
DATA BECKER ');
     writexy(20,4,'M E N Ú');
     writexy(20,5,'~~~~~~~');
     writexy(15,6,'1  Cambiar clave');
     writexy(15,8,'2  Consulta de clave');
     writexy(15,10,'3  Fin');
     writexy(15,13,'Su elección: ');
     readln(choice);
     if choice = 1 then begin;
      Cambiar_Clave;
      Guardar_Clave;
     end;
     if choice = 2 then begin;
      Consultar_Clave;
      ComprobarClave;
      ReaccionarClave;
     end;
  until choice = 3;
end;


begin;
  Menu;
end.
```

En la opción 1 puede introducir una nueva clave, en opción 2 se simula la consulta de clave. Tenga en cuenta la división de los diferentes tres procedimientos. En su programa no debería llamarlos en secuencia directa, sino de forma diseminada. Si

quiere emplear el tipo de clave 3, es importante no tener una sola clave, sino controlar una palabra determinada de una tabla de varias. Puede encontrar este método aplicado frecuentemente en juegos. Allí se consulta la palabra número tal-y-tal de una página determinada. Aparte de la palabra también se pueden consultar símbolos, colores, etc. El principio de consulta es el mismo, sólo los procedimientos de entrada y salida se modifican.

En el siguiente ejemplo podrá encontrar realizada este tipo de consulta. Bajo la opción 1 dispone de la posibilidad de introducir una lista de diez claves de acceso junto con un número de página ficticio. Bajo la opción 2 se puede introducir una de las claves, en un control simulado. El programa elige aleatoriamente una de las diez claves del archivo. En este ejemplo no se ha incluido una codificación por razones de espacio. Pero es muy aconsejable ya que de lo contrario sería muy fácil encontrar y modificar las palabras. Puede encontrar un ejemplo de codificaciones en el programa anteriormente presentado.

```
program Clave_Tipo3;
uses crt,design;

Type TipoClave = record
       Pagina : byte;
       text   : string[20];
     end;

VAR Claves : array[1..10] of TipoClave;
    Pwort  : TipoClave;
    clave  : string;
    ComprobarClave : boolean;
    Pw_Index : byte;

procedure Introducir_lista;
var li : integer;
begin;
 for li := 1 to 10 do begin;
  save_screen;
  Ventana(10,8,60,9,' Cambiar clave ',black,7);
  writexy(13,10,'Introducir la clave ');
  write(li,' de 10 de la lista');
  writexy(13,12,'Pagina de la clave en el manual: ');
  readln(Claves[li].Pagina);
  writexy(13,14,'Clave :');
  readln(Claves[li].text);
  restore_screen;
  textcolor(7);
  textbackground(black);
 end;
end;
```

```
end;

Procedure Guardar_Lista;
var pwf : file;
begin;
  assign(pwf,'Passtyp3.dat');
  rewrite(pwf,10*sizeof(TipoClave));
  blockwrite(pwf,Claves,1);
  close(pwf);
end;

procedure Cargar_Clave(Idx : byte);
var pwf : file;
begin;
  assign(pwf,'Passtyp3.dat');
  reset(pwf,1);
  seek(pwf,Idx*sizeof(TipoClave));
  blockread(pwf,Pwort,sizeof(TipoClave));
  close(pwf);
  save_screen;
  Ventana(10,8,45,7,'',black,7);
  writexy(12,10,'Introducir la clave de la página ');
  write(Pwort.Pagina);
  writexy(12,12,'Clave: ');
  readln(clave);
  restore_screen;
end;

procedure ComprobarClave;
begin;
  if clave = Pwort.text then
    ComprobarClave := true
  else
    ComprobarClave := false;
end;

procedure ReaccionarClave;
begin;
  save_screen;
  Ventana(10,8,40,7,'',black,7);
  If ComprobarClave then begin;
    writexy(13,11,'Clave correcta - Acceso permitido');
  end else begin;
    writexy(13,11,'Clave ERRONEA - ¡No hay acceso!');
  end;
  repeat until keypressed; readkey;
  restore_screen;
  textcolor(7);
  textbackground(black);
```

```
procedure menu;
var choice : byte;
begin;
  repeat
    clrscr;
    writexy(10,1,'Programa de ejemplo para el tipo de clave 3 (c)
''94 by DATA BECKER ');
    writexy(20,4,'M E N Ú');
    writexy(20,5,'~~~~~~~');
    writexy(15,6,'1  Introducir lista de claves');
    writexy(15,8,'2  Consulta de clave');
    writexy(15,10,'3  Fin');
    writexy(15,13,'Su elección: ');
    readln(choice);
    if choice = 1 then begin;
      Introducir_lista;
      Guardar_Lista;
    end;
    if choice = 2 then begin;
      Pw_Index := random(10)+1;
      Cargar_Clave(Pw_Index);
      ComprobarClave;
      ReaccionarClave;
    end;
  until choice = 3;
end;


begin;
  textcolor(7);
  textbackground(black);
  Menu;
end.
```

9.2 Programas de lenguajes avanzados a nivel de máquina

Ahora que ha conocido las diferentes posibilidades fundamentales de un control de claves y su realización en Pascal, vamos a dedicarnos a la programación orientada a la máquina. Sólo a nivel de la máquina se pueden realizar consultas realmente efectivas, ya que un programa en Pascal siempre se puede examinar mejor. Esto no depende de su forma de programar, sino de algunos procedimientos característicos que emplea el compilador de Pascal.

La estructura de un programa Pascal

Consideremos de momento este sencillo programa:

```
Program Little_Program;

uses crt;
VAR i : integer;
begin;
  clrscr;
  for i := 5 downto 1 do begin;
    gotoxy(10,6);
    writeln('Por favor espere ',i,' segundos');
    delay(1000);
  end;
end.
```

Si lo desensamblamos por ejemplo con el Turbo Debugger, se pueden reconocer muy bien algunas partes características del programa en Pascal. Al principio el programa llama los procedimientos de inicialización de las Units incluidas. En este caso son SYSTEM.TPU y CDT.TPU.

```
LITTLE_PROGRAM.7: begin;
  cs:001C 9A0000D862      call   62D8:0000
  cs:0021 9A0D007662      call   6276:000D
  cs:0026 55              push   bp
  cs:0027 89E5            mov    bp,sp
```

Después se llama el procedimiento *CLRSCR* de la Unit *CRT*. Si ha compilado el programa sin informaciones para un depurador externo, encontrará una dirección *Far* en las inicializaciones en vez del símbolo *CRT.CLRSCR*.

```
LITTLE_PROGRAM.8:  clrscr;
  cs:0029 9ACC017662      call   far CRT.CLRSCR
```

A continuación se inicializa el bucle. La variable *i* se encuentra como posición de memoria *[0052h]* en el segmento de datos. Se le asigna el valor *5*. En la primera ejecución del bucle el valor asignado es correcto, y el programa salta a la línea de programa Pascal *10*. En todas las demás ejecuciones se decrementa [*0052h*], es decir i, en 1.

```
LITTLE_PROGRAM.9:  for i := 5 downto 1 do begin;
  cs:002E C70652000500    mov    word ptr [0052], 0005
  cs:0034 EB04            jmp    LITTLE_PROGRAM.10 (003A)
  cs:0036 FF0E5200        dec    word ptr [0052]
```

En el bucle se sucede ahora la llamada del procedimiento gotoxy. Se le pasan las coordenadas x e y, a las que ha de saltar el procedimiento. Para ello se colocan en el stack los valores *000Ah* para la coordenada x y *0006h* para la coordenada y. A continuación se llama todo el procedimiento mediante un *Far Call*.

```
LITTLE_PROGRAM.10:  gotoxy(10, 6);
  cs:003A 6A0A          push   000A
  cs:003C 6A06          push   0006
  cs:003E 9A1F027662    call   far CRT.GOTOXY
```

En la siguiente línea se encuentra la visualización del mensaje mediante el procedimiento *writeln*. El procedimiento *writeln* posee la característica de que se le pueden pasar varios argumentos, es decir, que puede visualizar una cadena, así como una combinación de estas y variables. En nuestro ejemplo se ve claramente cómo lo traduce el compilador: para cada argumento se llama un procedimiento propio. Aquí se ha de distinguir entre tres tipos diferentes. Por una parta la salida de la cadena (String). Esto lo realiza el procedimiento *0615h* de la SYSTEM.TPU. Para ella se guarda, no la cadena a reproducir, sino su dirección en el stack. A continuación viene la representación del número *[0052h]*. Se copia a los registros *AX:DX* y se le pasa al procedimiento *06D9h* de la SYSTEM.TPU mediante el stack. Finalmente encontramos el procedimiento para generar el retorno de carro. Su inclusión representa la única diferencia con respecto al procedimiento *write*. Se puede encontrar en la posición *0582h* de la SYSTEM.TPU.

```
LITTLE_PROGRAM.11:  writeln('Por favor espere ',i,' segundos');
  cs:0043 BF6801          mov   di,0168
  cs:0046 1E              push  ds
  cs:0047 57              push  di
  cs:0048 BF0000          mov   di,0000
  cs:004B 0E              push  cs
  cs:004C 57              push  di
  cs:004D 6A00            push  0000
  cs:004F 9A1506D862      call  62D8:0615

  cs:0054 A15200          mov   ax,[0052]
  cs:0057 99              cwd
  cs:0058 52              push  dx
  cs:0059 50              push  ax
  cs:005A 6A00            push  0000
  cs:005C 9A9D06D862      call  62D8:069D

  cs:0061 BF1200          mov   di,0012
  cs:0064 0E              push  cs
  cs:0065 57              push  di
  cs:0066 6A00            push  0000
  cs:0068 9A1506D862      call  62D8:0615
  cs:006D 9A8205D862      call  62D8:0582
```

Después de visualizar el mensaje, se realiza la llamada del procedimiento *Delay* en el bucle. Antes de que se llame, se guarda en el stack el valor de *MS*, que ha de esperarse. En este caso con *03E8 (= 1000d) MS*.

```
LITTLE_PROGRAM.12:  delay(1000);
  cs:0072 68E803          push   03E8
  cs:0075 9AA8027662      call   far CRT.DELAY
```

Al final del bucle, que se señaliza mediante *end*, se comprueba si *[0052h] (i)* corresponde al valor de interrupción *1*. Si no es así, el procedimiento vuelve a saltar al principio del bucle y continúa allí con el decrementado de la variable *[0052h]*.

```
LITTLE_PROGRAM.13:  end;
  cs:007A 833E520001      cmp    word ptr [0052], 0001
  cs:007F 75B5            jne    0036
```

Si se ha alcanzado el valor destino, el programa ha llegado a la última línea del mismo, el *end*. Aquí se elimina primero el marco del stack. A continuación se borrará el registro *AX*, que contiene el valor resultante del programa y se llama el procedimiento de terminación *0116h* de la SYSTEM.TPU.

```
LITTLE_PROGRAM.14: end.
  cs:0081 C9              leave
  cs:0082 31C0            xor    ax, ax
  cs:0084 9A1601D862      call   62D8:0116
```

Ha visto que en este pequeño programa se pueden encontrar una gran cantidad de procedimientos característicos. Con su ayuda se puede identificar un archivo EXE con facilidad como generado por Turbo Pascal. Aparte de la inicialización de la Unit, a nosotros nos interesa especialmente el procedimiento *DELAY* de la Unit *CRT*. Echemos un vistazo a esta rutina:

```
CRT.DELAY
  cs:02A8 8BDC            mov    bx, sp
  cs:02AA 368B4F04        mov    cx, ss:[bx+04]
  cs:02AE E313            jcxz   02C3
  cs:02B0 8E064400        mov  es, [SYSTEM.SEG0040]
  cs:02B4 33FF            xor    di, di
  cs:02B6 268A1D          mov    bl, es:[di]
  cs:02B9 A16000          mov    ax,[0060]
  cs:02BC 33D2            xor    dx,dx
  cs:02BE E80500          call   02C6
  cs:02C1 E2F6            loop   02B9
  cs:02C3 CA0200          retf   0002
  cs:02C6 2D0100          sub    ax, 0001
  cs:02C9 83DA00          sbb    dx, 0000
```

```
cs:02CC 7205        jb    02D3
cs:02CE 263A1D      cmp   bl, es:[di]
cs:02D1 74F3        je    02C6
cs:02D3 C3          ret
```

Primero se carga el contador MS en CX. Es decir, que cuando en este lugar se encontrara

```
cs:02AA  B90100     mov   cx,0001
cs:02AD  90         nop
```

el bucle sólo se recorrería una vez. A continuación se recorre un bucle hasta que haya transcurrido el número especificado de MS.

La facilidad de comprensión del procedimiento DELAY nos indica que es lo que nosotros hemos de cuidar más: en los procedimientos importantes como bucles de espera y consultas de claves de acceso se debería prescindir de los procedimientos de que dispone Pascal. Se recomienda desarrollar soluciones propias, más especiales. Pero esto no significa necesariamente que se ha de volver a inventar la rueda de nuevo. De lo contrario, se pueden tomar como base los procedimientos existentes. Mediante una inversión de la secuencia de instrucciones y algunas pequeñas modificaciones se le puede tomar el pelo a más de un escáner de instrucciones.

Para nuestro procedimiento de espera queremos alejarnos completamente del procedimiento DELAY y desarrollar nuestra propia rutina de espera. Es equivalente al procedimiento DELAY, pero no es encontrada por ningún escáner de programas. La solución que presentamos aquí desvía la interrupción del temporizador a nuestra propia rutina. Para poder trabajar con exactitud, se reprograma la interrupción del temporizador a 1000 llamadas por segundo. Para una mejor comprensión se emplean las rutinas de la Unit DOS para el manejo de las interrupciones. En el capítulo 10 le mostraremos como puede pasar sin estas rutinas, para proteger además sus programas un poco más.

```
program esperar;
uses crt,dos;
var oldtimer : pointer;
    mscount  : longint;
    msready  : boolean;
    i        : integer;
procedure espera_int; interrupt;
begin;
  dec(mscount);
  if mscount = 0 then msready := true;
  port [$20] := $20;
```

```
end;
procedure espera(ms : longint);
begin;
 Getintvec(8,oldtimer);
 SetIntVec(8,@espera_int);
 asm
   cli
   mov dx,43h
   mov al,36h
   out dx,al
   sub dx,3
   mov al,169
   out dx,al
   mov al,4
   out dx,al
   sti
 end;
 msready := false;
 mscount := ms;
 repeat until msready;
 asm
   cli
   mov dx,43h
   mov al,36h
   out dx,al
   sub dx,3
   xor ax,ax
   out dx,al
   out dx,al
   sti
 end;
 SetIntVec(8,oldtimer);
end;
begin;
 clrscr;
 for i := 5 downto 1 do begin;
   gotoxy(10,6);
   writeln('Por favor, espere ',i,' segundos');
   warte(1000);
 end;
end.
```

9.3 Una clave de acceso protegida

El método aquí presentado de clave de acceso protegida seguramente no es la mejor protección posible, pero al menos podrá detener a muchos Hackers aficionados a desmontar la protección. La versión presentada guarda las claves de acce-

so y las indicaciones de página como constantes. Esto naturalmente lo debería sustituir al emplearlo en sus propios programas por uno de los métodos anteriormente descritos. Sin embargo, aquí tiene sentido para una mejor comprensión.

El programa no contiene ningún escollo insalvable, pero a cambio sí una serie de pequeños trucos. Todo comienza con la instrucción mem del principio. No está implementada por que realmente necesitemos tanta memoria, sino para que el programa no pueda ser depurado por el Turbo Debugger normal. Esta trampa se puede desconectar reduciendo el valor del offset 0ah en la cabecera EXE, por ejemplo a 0d430h para 200.000 bytes. Con ello se puede depurar el programa con el Turbo Debugger normal, pero al intruso le esperan otros pequeños trucos agradables. El bucle principal que controla la clave de acceso está en ensamblador. En ella desconectamos el teclado por vía hardware. A continuación se dispara un int 3h. Esta interrupción no tiene ningún efecto si no hay ningún depurador instalado. Pero si en segundo plano está trabajando el TD, se interrumpe el programa en este lugar. Esto también suele ser el final para el TD normal, ya que el programa ya no se puede manejar a causa del teclado desconectado. Como mucho queda la alternativa del ratón, pero también la interrupción 33h de este se puede desviar...

Si se atravesara esta barrera, el Hacker se encontrará con el truco PIQ que le presentaremos en el siguiente capítulo. Los saltos condicionales completamente innecesarios ayudan a crear un poco más de confusión. Si se saltaran todas estas barreras, sólo nos queda esperar que nuestro Hacker lo deje por imposible cuando se de cuenta de que no sólo ha de desmontar toda esta paranoia una sola vez, sino unas cuantas más. Si tampoco esto le impide continuar, ya no se trata de un Hacker Junior, sino de un Freak acostumbrado. Y contra estos no hay nada que hacer. Pero volvamos a nuestro punto de partida: la asignación de memoria. Si nuestro Hacker no sabe como desmontarla, hará servir el TD386. Este ofrece, aparte de necesitar poca memoria, la ventaja de la virtualización completa del PC. Esto significa para nosotros que los trucos de las desviaciones de interrupciones y la desconexión del teclado se realizan en la máquina virtual y no tienen ningún efecto sobre el TD. Pero hay un pequeño detalle que el TD 386 no soporta en cualquier caso: el Protected Mode. Así que conmutamos en diferentes puntos del programa aleatoriamente al Protected Mode y de vuelta. Esto no afecta en absoluto a nuestro programa, pero el TD 386 finaliza con un error de Excepción. Una última barrera la representan las llamadas indirectas de procedimientos. Los procedimientos del programa den Pascal no se llaman por su verdadero nombre, sino a través de un puntero que contiene el nombre del procedimiento. Esto complica la legibilidad del programa un poco. Aquí tiene el código del programa de claves de acceso:

```
{$F+}
{$M $4000,500000,650000}
program Consulta_clave;

uses crt,design;
const Claves : array[1..10] of string =
        ('Marcombo','Inspire','PC Underground','Soundblaster',
         'Demos','Super','Vengeance','Dynamite','Cerveza','Casa');
      Pw_Pages : array[1..10] of word =
        (17,3,29,43,12,21,4,9,13,30);

Var pw_nr : byte;
    Ciclos_restantes : byte;
    Clave_correcta : word;
    Nueva_clave : string;
    PElegir_nueva_clave : pointer;
    PDibujar_cuadro_entrada : pointer;
    PConsultar_clave : pointer;
    PDetener_sistema : pointer;
    Var_innecesaria1 : word;
    Var_innecesaria2 : word;

{$L Pwmodul}
procedure Bucle_Consulta; far; external;

procedure Elegir_nueva_clave;
begin;
  pw_nr := random(10)+1;
  Var_innecesaria1 := 1;
  Var_innecesaria2 := 2;
end;

procedure Dibujar_cuadro_entrada;
var pws : string;
begin;
  str(Pw_Pages[pw_nr]:2,pws);
  asm int 3; end;
  Ventana(20,10,40,4,'Introducir clave en pág. '+pws,black,7);
  Var_innecesaria1 := 1;
  Var_innecesaria2 := 2;
  gotoxy(23,12);
end;

procedure Consultar_Clave;
begin;
  readNueva_clave);
  Var_innecesaria1 := 1;
  Var_innecesaria2 := 2;
  if Nueva_clave = Claves[pw_nr] then
```

```
    Clave_correcta := 1
  else
    Clave_correcta := 0;
end;

procedure Detener_sistema;
begin;
  textbackground(black);
  textcolor(7);
  clrscr;
  writeln('Mejor que hubiéramos comprado un original, eh? ...');
  halt(0);
end;


procedure Main_Programm;
begin;
  textbackground(black);
  textcolor(7);
  clrscr;
  gotoxy(20,12);
  writeln('Bienvenido en el programa principal !');
  gotoxy(20,22);
  write('<Enter> para terminar ... ');
  readln;
  halt(0);
end;

begin;
  textbackground(black);
  textcolor(7);
  clrscr;
  Ciclos_restantes := 57;
  PElegir_nueva_clave := @Elegir_nueva_clave;
  PDibujar_cuadro_entrada    := @Dibujar_cuadro_entrada;
  PConsultar_clave       := @Consultar_Clave;
  PDetener_sistema         := @Detener_sistema;
  randomize;
  Bucle_Consulta;
end.

.386p
.MODEL TPASCAL

keyb_off macro
  push ax
  in  al,21h
  or  al,02
  out 21h,al
```

```
  pop ax
endm

keyb_on macro
  push ax
  in  al,21h
  and al,0Fdh
  out 21h,al
  pop ax
endm

.DATA
 extrn Ciclos_restantes
 extrn PElegir_nueva_Clave        : dword
 extrn Pdibujar_cuadro_entrada    : dword
 extrn PPreguntar_Clave           : dword
 extrn PDetener_sistema           : dword
 extrn Clave_correcta             : byte
 extrn Var_innecesaria1           : word
 extrn Var_innecesaria2           : word

.CODE
extrn Main_Programm          : far

public Bucle_control

Bucle_control proc pascal
  keyb_off

 ;PIQ - Trick
  int 3
  mov  cs:word ptr [@int_21_func1],4CB4h  ; función terminar programa
@int_21_func1:
  mov  ah,30h                             ; función obtener versión DOS
  int 21h

@Loop_control:
  keyb_off

  call dword ptr PElegir_nueva_Clave
  cmp  Var_innecesaria1,5
  jbe  @Etiq_innecesaria1a

 :PIQ - Trick
  int 3
  mov  cs:word ptr [@Etiq_innecesaria2],4CB4h  ; función fin programa
@Etiq_innecesaria2:
  mov  ah,30h                             ; función obtener versión DOS
  int 21h
```

```
  mov  cs:word ptr [@Etiq_innecesaria2],30B4h  ; función fin programa

  call dword ptr Pdibujar_cuadro_entrada
  jmp  @Etiq_innecesaria1b

@Etiq_innecesaria1a:
 ;PIQ - Trick
  int 3
  mov  cs:word ptr [@Etiq_innecesaria2],4CB4h  ; función fin programa
@Etiq_innecesaria2a:
  mov  ah,30h                                  ; función obtener versión DOS
  int 21h
  mov  cs:word ptr [@Etiq_innecesaria2a],30B4h ; función fin programa

  call dword ptr Pdibujar_cuadro_entrada
@Etiq_innecesaria1b:
  keyb_on


  cmp  Var_innecesaria2,10
  jbe  @Etiq_innecesaria2a

  dec  byte ptr Ciclos_restantes

; Protected MODE Trick
   pusha
   cli                          ; apagar interrupciones
   mov eax,cr0                  ; pasar al Protected-Mode
   or eax,1
   mov cr0,eax
   jmp PROTECTION_ENABLED       ; borrar Executionpipe
PROTECTION_ENABLED:
   and al,0FEh                  ; pasar de nuevo al modo real
   mov cr0,eax                  ; no resetear CPU
   jmp PROTECTION_DISABLED      ; borrar Executionpipe
PROTECTION_DISABLED:
   sti                          ; activar interrupciones
   popa

  call dword ptr PPreguntar_Clave
  jmp  @Etiq_innecesaria2b

@Etiq_innecesaria2a:
  dec  byte ptr Ciclos_restantes

; Protected MODE Trick
   pusha
   cli                          ; apagar interrupciones
   mov eax,cr0                  ; pasar al Protected Mode
```

```
   or eax,1
   mov cr0,eax
   jmp PROTECTION_ENABLED2a     ; borrar Executionpipe
PROTECTION_ENABLED2a:
   and al,0FEh                  ; volver al modo real
   mov cr0,eax                  ; no resetear CPU
   jmp PROTECTION_DISABLED2a    ; borrar Executionpipe
PROTECTION_DISABLED2a:
   sti                          ; activar interrupciones
   popa
  call dword ptr PPreguntar_Clave

@Etiq_innecesaria2b:

  cmp  byte ptr Clave_correcta,1
  je   @Consulta_OK
  jmp  @Consulta_no_OK
@Consulta_OK:

  call Main_Programm
@Consulta_no_OK:

  cmp byte ptr Ciclos_restantes,54
  ja  @Loop_control
  call dword ptr PDetener_sistema
  ret
Bucle_control endp

END
```

10. Proteja sus conocimientos - trucos anti-debugging

La protección probablemente más segura contra copias piratas es el empleo de un Dongle. Sin embargo, con los Dongles ocurre como con todo: cuanto mejor es la calidad, tanto más alto es el precio. Si por ejemplo quiere proteger un juego, este camino está vedado. El Dongle valdría un múltiple del juego. Y un Dongle de 800 ptas no trae otra cosa que una seguridad óptica. Otra posibilidad de relativa seguridad es el empleo de un CD-ROM. Es menos importante el hecho de que no se pueda escribir en él, sino que alberga tal cantidad de datos que la copia resulta impracticable. Así que si el programa no se copia de forma profesional, el CD ofrece una cierta seguridad. Muchos usuarios se pensarán si liberan 200 MBytes en su disco, sólo para un programa (aunque los nuevos compiladores de Borland comienzan a tomar estas dimensiones...). Si además reproduce el sonido directamente desde el CD, ya ha fastidiado a muchos copiones, aunque puede perjudicarse a sí mismo. La unidad CD-ROM sigue siendo un dispositivo en el que todavía se ahorra. Esto significa que no puede llegar a todos los usuarios que no disponen de unidad CD-ROM.

Así que si estas dos posibilidades no son aplicables para usted, sólo le queda un camino: ha de incluir una protección de copia en su programa. Y esta debería estar asegurada. Pero una cosa la hemos de decir desde el principio: no existe ninguna protección de copia que no se pueda saltar. Pero se le puede facilitar la vida a los ladrones de datos, o complicarsela bastante. Especialmente ahora, en una época de criminalidad económica ascendente, los trucos anti-debugging que presentamos a continuación, ganan cada vez más en importancia. Al fin y al cabo a nadie le gusta que le roben su trabajo de investigación. El mejor ejemplo para ello podría ser Stack Electronics, cuyo código fue robado simplemente por Microsoft para su DOS 6.0. Y al revés. Las consecuencias jurídicas deberían ser conocidas. Los trucos que aquí presentamos seguramente no le protegerán de un Hacker profesional, pero representan una buena pantalla para un rey del Debugging junior. Puede encontrar estos trucos en muchos programas profesionales.

10.1 Inspeccione programas

Para inspeccionar programas hay toda una serie de utilidades. Algunos de los mejores se los presentaremos a continuación. Bajo utilidades entenderá probablemente programa universales como las PC Tools o las Norton Utilities. Pero no ha-

blaremos de este grupo de programas. Sino que presentaremos algunos más útiles que le simplificarán la vida con el ordenador. A continuación conoceremos un representante de las tres grandes categorías de los descompresores, depuradores (debugger) y editores hexadecimales.

Abajo con el camuflaje - programas de descompresión

Mediante programas como PKLITE™ puede comprimir archivos EXE y COM, permaneciendo sin embargo ejecutables. Esto tiene la ventaja de que ya no ocupan tanto espacio en el disco. El tiempo que necesitan para descomprimirse no se nota en absoluto. Pero también albergan un gran problema. Un programa puede estar afectado por un virus. Si este programa ahora se comprime, ¡el virus no podrá ser reconocido por ningún detector anti-virus!

También cuando quiere ver el modo de funcionamiento del programa (saludos de la ley de la UE), es mejor si dispone del programa en su forma directamente ejecutable. Y para deshacer este proceso de compresión, existen los programas descompresores. La capacidad de estos programas (normalmente Shareware) suele ser sorprendente. Casi no hay ningún programa que esté seguro ante ellos. Le presentamos el programa que actualmente es el mejor descompresor. Se trata del programa UNP (Unpack), que durante la creación de este libro estaba disponible en su versión 3.12. UNP está en disposición de descomprimir todos los programas comprimidos con cualquier compresor habitual. Lo puede llamar de la siguiente forma:

```
UNP Comando [Opciones] Archivo fuente [Archivo destino]
```

Puede sustituir *Comando* por lo siguiente:

e - descomprimir archivo (expand compressed)

Con este comando UNP descomprime el archivo especificado. Si no indica ningún nombre de archivo, se descomprimen todos los archivos del directorio actual. Si no indica ningún comando, UNP adopta automáticamente el comando *e*.

c - Conversión en archivo COM (convert to com)

Hay algunos archivos EXE, que pueden ser convertidos en archivos COM. ¡Estos archivos no deben contener segmento de datos! Este tipo de archivos EXE pueden ser convertidos con esta opción en un archivo COM. Esto tiene la ventaja de que los archivos serán más pequeños.

i - Visualizar información (info only)

Si sólo quiere informaciones sobre el programa a analizar (¿Estará el archivo comprimido?), llame a UNP con el comando *i*. Obtendrá todas las informaciones que hubiera mostrado el comando *e*. Pero el archivo no se descomprime.

l - Cargar y guardar (load & save)

Esta opción es útil cuando quiere eliminar datos de cabecera innecesarios del programa. El programa comprimido se carga en la memoria y se guarda de nuevo de forma comprimida. Este comando es interesante junto con la opciones *-h* y *-k*.

s - Buscar archivos comprimidos (search for compressed files)

Mediante esta opción puede ver la lista de todos los archivos comprimidos de un directorio determinado. Se comprueban todos los archivos que corresponden al comodín especificado. Se listan todos los archivos comprimidos y UNP visualiza informaciones sobre qué compresor compactó los archivos.

x - Conversión en un archivo EXE (convert to EXE)

Mediante esta opción se puede convertir un archivo COM en EXE. Esto tiene sentido, ya que algunos compresores sólo pueden compactar archivos EXE. Si quiere comprimir su archivo COM, conviértalo primero en EXE con UNP, y comprímalo después. Puede realizar otros ajustes a través de las opciones de UNP. En la versión 3.12 existen las siguientes posibilidades:

-? - Mostrar ayuda (help)

Al introducir esta opción, UNP le muestra una lista de todos los comandos y opciones. Todos los demás comandos y opciones serán ignorados.

-a - Intentar de nuevo automáticamente (automatic retry)

Hay algunos archivos que fueron comprimidos más de una vez, o que fueron "protegidos" por otros programas como el paquete Anti-virus de Central Point. Si especifica esta opción, se comprueba automáticamente después de la descompresión del archivo si este está realmente desempaquetado. De lo contrario el proceso comienza de nuevo.

-b - Crear archivo de Backup (make .BAK-file)

Queremos aconsejarle emplear esta opción en cualquier caso, si su espacio en el disco lo permite. Al fin y al cabo podría ser que... ¡su archivo desempaquetado ya no funcione! Es bastante tranquilizante disponer de una copia de seguridad con la extensión .BAK.

-c - Preguntar antes de descomprimir (ask for confirmation)

Si especifica esta opción, (importante junto con el uso de comodines), UNP le pregunta cada vez antes de desempaquetar un archivo.

-h - Eliminar datos de cabecera (remove irrelevant header data)

Debería emplear esta opción, si ha comprimido un programa. Con ella puede eliminar los "innecesarios" identificadores de Copyright del compresor, que no son necesarios para la ejecución.

-i - No interceptar la Interrupción 21h (do not intercept 21h calls)

Normalmente UNP emplea la interrupción *21h* del DOS para comprobar su el programa funciona perfectamente. Pero hay algunos simpáticos TSR (como por ejemplo Turbo Debugger), con los que se pueden producir problemas y cuelgues de programa. Mediante esta opción se puede desconectar esta comprobación interna.

-k - Signatura PKLite (pklite signature)

Mediante esta opción se puede eliminar o añadir la signatura de *PKLite*. Si quiere comprimir un programa, naturalmente tiene sentido el eliminarla (de nuevo bytes innecesarios).

Mediante *-k* elimina la signatura. Esto también se aplica si sólo introduce *-k*.

Mediante *-k+* se le añade la signatura PKLite al programa. Esto sólo tiene sentido si se quiere asegurar de que sus programas sean fáciles de descomprimir. UNP adopta esta opción automáticamente, de modo que también la puede obviar.

Mediante *-k?* provoca que UNP le pregunte cada vez cuando encuentre un signatura.

-l - Emplear siempre Loadfix (always use loadfix)

Unpack normalmente no llena los primeros 64 KBytes de la memoria principal, para poder procesar archivos mayores. Pero si precisamente se necesita esto (¡archivos comprimidos por EXEPACK!), UNP carga el programa de nuevo, y precisa-

mente en esta zona. Si ahora quiere descomprimir varios archivos, debería especificar esta opción, para que el programa no se haya de cargar de nuevo cada vez.

-o - Sobrescribir el archivo de salida (overwrite output file)
Si siempre quiere sobrescribir el archivo de salida, si existe, debería especificar esta opción. De lo contrario UNP pregunta cada vez si debe sobrescribir el archivo o no.

-p - Alinear datos de cabecera a la página (align header data on a page)
Mediante esta opción puede aumentar la cabecera *EXE* de forma que se adapte exactamente a una página de 512 bytes. Esto debe provocar que el archivo EXE se cargue más rápidamente.

-r - Eliminar datos Overlay (remove overlay data)
En algunos archivos de programa se añaden datos de overlay. Mediante esta opción puede eliminar estos datos. Esto suele ser poco útil, ya que suelen contener datos importantes.

-u - Actualizar fecha y hora del archivo (update file date / time)
Normalmente UNP suele emplear la fecha y la hora de archivo del original. Mediante esta opción obliga a UNP para que emplee la fecha y la hora actuales.

-v - Otras informaciones (verbose)
Mediante esta opción puede obtener más opciones para el proceso de descompresión. Estas informaciones normalmente no se necesitan. Si quiere descomprimir por ejemplo el archivo HOLA.EXE, debería introducir lo siguiente:

```
UNP e -a -o HOLA.EXE
```

Si quiere eliminar todos los datos innecesarios de un archivo comprimido, indique esto:

```
UNP l -k- -h HOLA.EXE
```

Paso a paso - Turbo Debugger

El Turbo Debugger suministrado con Borland C++ y Pascal, es uno de los mejores depuradores de los que se puede disponer actualmente. Con el puede procesar paso a paso tanto sus códigos fuente, como archivo EXE y COM terminados. Con el Turbo Debugger también puede fijar los llamados Breakpoints. Puede dejar que

su programa se ejecute hasta ese punto, y desde allí procesarlo paso a paso. A continuación le explicaremos cómo puede analizar un programa con esta potente herramienta y cómo puede vigilar determinadas variables. Informaciones más profundas y específicas, como por ejemplo para la depuración orientada a objetos, las podrá encontrar en el estupendo manual.

La llamada del Turbo Debugger

Primero ha de decidir qué versión del Turbo Debugger quiere emplear. Puede usar el Turbo Debugger normal. Este funciona con cualquier administrador de memoria pero también necesita mucha memoria. Como alternativa también puede emplear el TD386. Para este depurador ha de incluir la línea

```
Device = TD386.sys
```

en su CONFIG.SYS. El controlador no se lleva bien con el EMM386 o QEMM. Así que si su programa necesita memoria EMS, no podrá emplear este debugger. De lo contrario, ofrece la ventaja de que puede depurar programas muy grandes y que usen mucha memoria con él. Así que si puede prescindir de EMS, el TD386 es la elección correcta en cualquier caso.

Los parámetros de línea de comando

Puede configurar individualmente el Turbo Debugger mediante parámetros en la línea de comando en cada llamada. Pero hay una serie de opciones que se necesitan cada vez. Estas se pueden guardar en un archivo de configuración. Y este se puede crear con el Turbo Debugger bajo OPTION - SAVE OPTIONS. Debería darle un nombre individual, que pueda encontrar con facilidad, como por ejemplo *miconfg.td*. Mediante el parámetro de línea de comandos *-c* puede cargar el archivo directamente cada vez que se ejecuta el Turbo Debugger. La llamada podría tener este aspecto

```
TD386 -cMICONFG.TD ElPrg
```

donde *ElPrg* representa al programa a depurar. Puede completar o modificar las opciones que aquí hemos indicado mediante otros parámetros de línea de comandos. Dispone de:

-d - Conmutación de pantalla, -do - emplear dos pantallas

Si aún posee una de las viejas tarjetas Hercules con monitor, puede indicarle al debugger que muestre el programa a depurar junto con él mismo en el monitor

Hercules, y que realice las representaciones gráficas en la pantalla VGA. Esto tiene la ventaja de que siempre puede ver exactamente qué es lo que su programa visualiza en el monitor en cada momento.

-dp - Page-Flipping (especialmente para modo de texto)

Al especificar esta opción, el Turbo Debugger trabaja con dos pantallas virtuales. pero esto sólo funciona en el modo de texto, y sólo si el programa a depurar no emplea por si mismo varias páginas de pantalla o juega con la dirección inicial de la RAM de vídeo. Si depura programas en modo gráfico, puede obtener fácilmente errores gráficos si conmuta entre la pantalla gráfica y la del debugger. Pero habitualmente estos errores son soportables. Ya que este modo es el más rápido, es el preferible.

-ds - Screen-swapping

Si elige este modo, el Turbo Debugger guarda el contenido de la pantalla del programa a depurar en un buffer, antes de restaurar la pantalla del debugger, y lo restaura después de las entradas de usuario. Este swapping necesita algo de tiempo y es bastante pesado si depura un programa paso a paso. Sólo seleccione este modo si lo necesita urgentemente.

-k - Grabación de teclado

Esta opción es bastante útil. Mediante ella se graban todas las entradas del usuario en el programa a depurar así como en el debugger. De esta forma puede ir rápidamente a un lugar determinado del programa, si ha depurado "demasiado lejos". Las introducciones de usuario se guardan en un archivo.

-l - Iniciar en modo ensamblador

Si ejecuta el debugger en el modo ensamblador, no obtiene el código fuente (opcional) durante el inicio, sino la ventana de la CPU. Los procesos realizados habitualmente durante la inicialización no se ejecutan directamente, sino que se puede observar el proceso de carga del programa.

-m - Determinar tamaño del Heap

El Turbo Debugger emplea un Heap para propósitos internos, que habitualmente mide 18 KBytes. Si entra en una falta aguda de memoria, puede reducir el tamaño del Heap a emplear hasta 7 KBytes. El tamaño de la memoria a usar se ha de indicar directamente después de -memoria en KBytes.

-v - Gestión de la memoria de pantalla, -vg - Guardar memoria de pantalla

Mediante esta opción se guardan siempre los 8 KBytes completos de memoria de pantalla. A pesar de que de esta forma se gastan 8 KBytes adicionales de memoria de pantalla, podrá depurar programas que usan esta memoria y cuya visualización normalmente sería destruida.

-vn - Sin modo de 50 líneas

Si está seguro de que no emplea el modo de 50 líneas, puede emplear este parámetro. De esta forma puede ahorrar algo de memoria.

-vp - Guardar paleta de colores

Algunos programas modifican la paleta VGA. Para estos programas debería emplear -vp. Con ello se guarda la paleta de color y siempre dispondrá de los colores adecuados al depurar el programa.

-y - Determinar tamaño de overlay del TD

El Turbo Debugger descarga todas las partes no usadas del programa en un Overlay. Según la memoria que necesite el programa a depurar, puede adaptar el tamaño de este Overlay. Si su programa necesita extremadamente mucha memoria, seleccione mediante *-y20* el requerimiento mínimo. Ahora tiene mucha memoria libre, pero habrá de pagarla con los tiempos de carga correspondientes. Si el programa a depurar necesita poca memoria, puede seleccionar el empleo máximo de la memoria de 200 KBytes mediante *-y200*. De esta forma se mantiene al debugger completo en la memoria, y puede trabajar de forma más rápida.

Búsqueda con el Turbo Debugger

Para nosotros no es tan interesante, sobre todo en relación a la programación de entrenadores (apartado 10.5), cómo se depura un programa con el Turbo Debugger del que disponemos del código fuente. Sino que nos interesa cómo podemos buscar determinadas variables o instrucciones en el programa que analizamos. Esto está permitido según las leyes de la UE, sólo que no puede realizar modificaciones en el archivo COM o EXE u otro perteneciente al programa. Cómo ha de proceder para poder encontrar por ejemplo las variables de "vidas" y "puntos" en un archivo EXE, se lo mostraremos a continuación. Para ello empleamos como base este pequeño programa en Pascal:

```
{$A+,B-,D-,E+,F+,G+,I+,L-,N-,O-,P-,Q-,R-,S+,T-,V+,X+,Y-}
{$M 16384,0,655360}
program show_how_to_debug;
uses crt;
```

```
Var inputvar : word;
    vidas   : byte;
    factor : real;

Procedure Inic_Variables;
begin;
  vidas  := 4;
  factor := 0.27
end;

Procedure Entrada_usuario;
begin;
  textcolor(14);
  gotoxy(10,3);
  write('
  gotoxy(10,3);
  write('Introduzca su nuevo valor de prueba : ');
  readln(inputvar);
end;
Procedure Evaluación_SinSentido;
begin;
  if (inputvar * factor) < 10 then
    dec(vidas);
end;

Procedure Escribir_estado;
begin;
  textcolor(15);
  gotoxy(10,10);
  write('Entrada : ',inputvar:5,'      ==>    Vidas : ',vidas:3);
end;
begin;
  clrscr;
  Inic_Variables;
  repeat
    Entrada_usuario;
    Evaluación_SinSentido;
    Escribir_estado;
  until vidas = 0;
end.
```

El programa lee un número del usuario. Si es menor de 38 se le resta una "vida". Esto se repite hasta que no queden vidas. Después de haber compilado el programa (lo puede encontrar en el CD adjunto), puede ejecutar el Turbo Debugger. Introduzca

```
TD td_test
```

y obtendrá la siguiente pantalla:

```
File  Edit  View  Run  Breakpoints  Data  Options  Window  Help                READY
cs:01B7▶9A00008257      call   5782:0000
cs:01BC 9A0D002057      call   5720:000D
cs:01C1 55              push   bp
cs:01C2 89E5            mov    bp,sp
cs:01C4 31C0            xor    ax,ax
cs:01C6 9ACD028257      call   5782:02CD
cs:01CB 9ACC012057      call   5720:01CC
cs:01D0 0E              push   cs
cs:01D1 E82CFE          call   0000
cs:01D4 0E              push   cs
cs:01D5 E8B1FE          call   0089
cs:01D8 0E              push   cs
cs:01D9 E81FFF          call   00FB
cs:01DC 0E              push   cs
cs:01DD E876FF          call   0156

ax 0000   c=0
bx 0000   z=0
cx 0000   s=0
dx 0000   o=0
si 0000   p=0
di 0000   a=0
bp 0000   i=1
sp 4000   d=0
ds 56F1
es 56F1
ss 5875
cs 5701
ip 01B7

ds:0000 CD 20 FF 9F 00 9A F0 FE ═ ƒ U═■
ds:0008 1D F0 E4 01 CD 23 AE 01 ↔≡Σ╦═#«╝
ds:0010 CD 23 B0 02 28 1E 96 0F ═#Ç╗(▲Ü▲
ds:0018 01 01 01 00 02 FF FF FF ☺☺☺ ■
ds:0020 FF FF FF FF FF FF FF FF

ss:4002 0000
ss:4000▶0000
ss:3FFE 0000
ss:3FFC 0000
ss:3FFA 0000

F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figura 20: Programa cargado

Ahora repase el programa paso a paso con <F8>. Irá encontrando las llamadas de los diferentes subprocedimientos. Intente averiguar que hacen estos procedimientos. Se dará cuenta de que se encuentra en un bucle, en el que primero se obtiene una entrada de usuario. El significado del siguiente procedimiento no se puede ver a primera vista. El tercero finalmente visualiza el valor introducir y las vidas restantes en la pantalla. A continuación viene una comparación muy interesante. Se comprueba si la variable [*0054*] tiene el valor *0*. Si no, se recorre el bucle de nuevo.

```
File  Edit  View  Run  Breakpoints  Data  Options  Window  Help                READY
cs:01DD E876FF          call   0156
cs:01E0▶803E540000      cmp    byte ptr [0054],00
cs:01E5 75ED            jne    01D4
cs:01E7 C9              leave
cs:01E8 31C0            xor    ax,ax
cs:01EA 9A16010257      call   5782:0116
cs:01EF 005589          add    [di-77],dl
cs:01F2 E5DB            in     ax,DB
cs:01F4 FF00            inc    word ptr [bx+si]
cs:01F6 9A16010257      call   5782:0116
cs:01FB 5D              pop    bp
cs:01FC C3              ret
cs:01FD 55              push   bp
cs:01FE 89E5            mov    bp,sp
cs:0200 E82800          call   022B

ax 0000   c=0
bx 0F00   z=1
cx 0300   s=0
dx 03D5   o=0
si 0217   p=1
di 0217   a=0
bp 3FFE   i=1
sp 3FFE   d=0
ds 5849
es 5849
ss 5875
cs 5701
ip 01E0

56F1:0000 CD 20 FF 9F 00 9A F0 FE ═ ƒ U═■
56F1:0008 1D F0 E4 01 CD 23 AE 01 ↔≡Σ╦═#«╝
56F1:0010 CD 23 B0 02 28 1E 96 0F ═#Ç╗(▲Ü▲
56F1:0018 01 01 01 00 02 FF FF FF ☺☺☺ ■
56F1:0020 FF FF FF FF FF FF FF FF

ss:4006 0000
ss:4004 0000
ss:4002 0000
ss:4000 0000
ss:3FFE▶0000

F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figura 21: Comprobar las variables

Deberíamos prestarle algo más de atención a esta variable, ya que parece ser importante. De modo que la incluimos en la ventana Watches, en la que se pueden comprobar determinadas zonas de memoria.

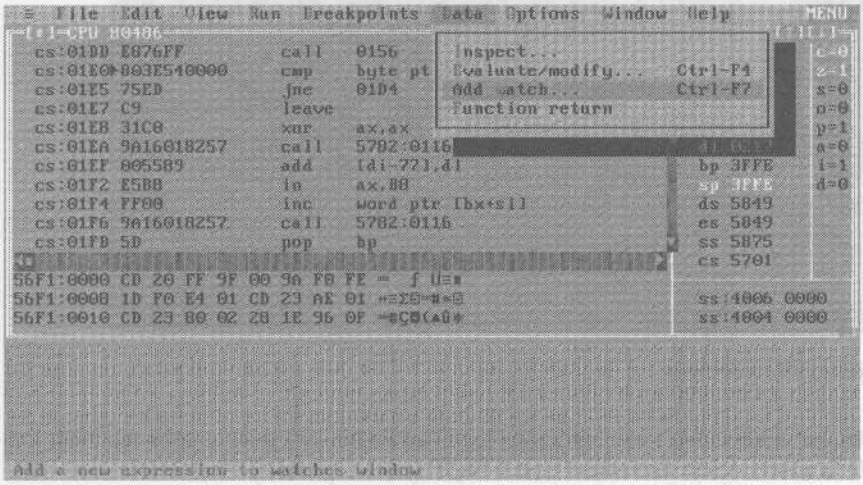

Figura 22: Crear variable Watch

Ahora seguimos procesando el programa. Se dará cuenta de que la variable tiene el mismo valor que la mostrada en el programa bajo *Vidas*. Así que se puede suponer con facilidad que realmente se trata de la variable para *Vidas*.

También el significado del segundo procedimiento, cuya tarea antes no estaba clara, se entiende ahora. Si se introduce un valor menor de 38, se decrementa la variable.

Ahora ya sabemos donde podemos encontrar al "malo" que nos descuenta las *Vidas*. Cuando pase la próxima vez por el segundo procedimiento, no lo pase de largo, sino que procéselo paso a paso.

Para ello pulse, cuando se encuentre sobre el procedimiento, la tecla [F7]. Ahora se encuentra en el subprocedimiento. Vaya paso a paso con [F8]. Casi al final del procedimiento encontrará la instrucción que descuenta las vidas, es decir

```
dec byte ptr[0054]
```

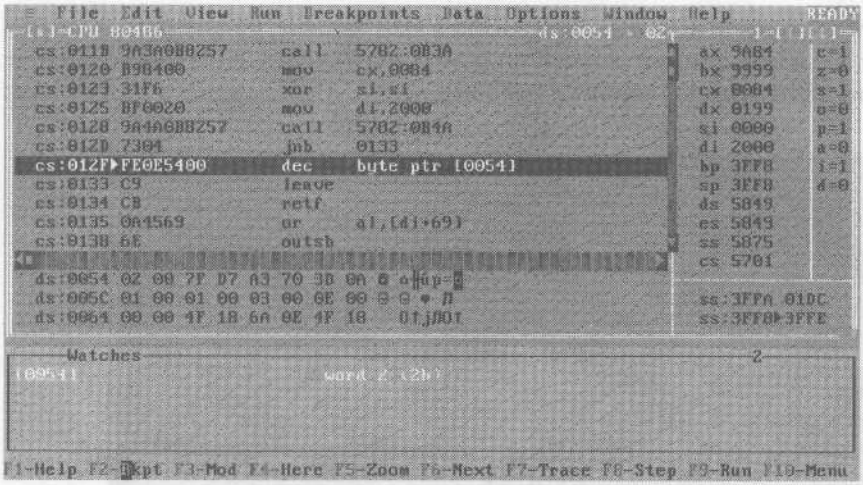

Figura 23: Instrucción encontrada

Para controlar si las vidas realmente se decrementan aquí, puede sobreescribir los 5 bytes de la instrucción *decrement* con 5 NOP (non Operate). Para introducir otra instrucción, pulse Space e introduzca la instrucción. Tenga en cuenta que la longitud de la instrucción antigua ha de coincidir con la nueva.

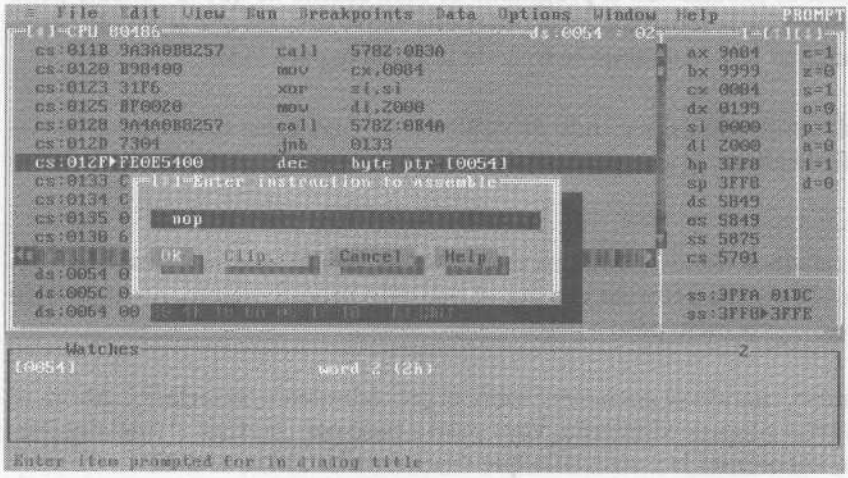

Figura 24: Introducir nuevas instrucciones

¡Atención! Incluso la colocación parcial de NOP está al borde de la legalidad. Al menos se dará cuenta de que la variable *[0054]* realmente ha sido la variable buscada de las *Vidas*. Si ahora escribe un entrenador, bien ha de cambiar la posición de

memoria en la que se guarda la variable, o bien la posición en la RAM de su ordenador en la que se decrementan las vidas. Aviso: ¡Fuera las manos del archivo EXE! No se le ocurra buscar el código de programa del DEC con un editor hexadecimal y sustituirlo por un NOP. ¡ESTO ESTÁ PROHIBIDO Y PERSEGUIDO POR LA LEY!

El Turbo Debugger le ofrece, aparte de los Watches y la posibilidad de procesar su programa paso a paso, toda una serie de otras funciones útiles. Esto incluye la posibilidad de detener la ejecución del programa en determinados puntos o ejecutar el programa hasta un lugar, así como muy buenas funciones de búsqueda y salto.

Fijar Breakpoint <F2>

Mediante la tecla [F2] puede fijar un así llamado *Breakpoint*. Esto significa que la ejecución del programa se interrumpe y el Turbo Debugger vuelve a tomar "las riendas" cuando se alcanza un punto determinado en el programa. Vaya el lugar del programa en el que quiere instalar semejante punto de parada y pulse allí la tecla [F2].

Colocar Breakpoint en un lugar determinado <Alt>+<F2>

Mediante [Alt] + [F2] tiene la posibilidad de colocar el punto de parada en un lugar determinado de la memoria. Para ello ha de introducir el segmento y la posición de offset del Breakpoint deseado. No necesariamente ha de encontrarse en el programa actual, sino que también puede estar en cualquier lugar de la memoria. Esto es muy útil si quiere depurar un controlador o una interrupción. Pero entonces debería saber exactamente lo que hace.

Ejecutar programa <F9>

Para iniciar el programa cargado, simplemente pulse la tecla [F9]. Habitualmente puede interrumpir el programa a depurar en el lugar "crítico" mediante [Ctrl] + [Break] para luego procesarlo paso a paso.

Ejecutar hasta la posición actual <F4>

Si se encuentra en un lugar determinado del programa y quiere volver a ejecutarlo desde el principio hasta allí mismo, no necesita depurarlo todo otra vez paso a paso hasta ese punto. Tiene la posibilidad de resetearlo mediante [Ctrl] + [F2] y dejar que se ejecute hasta la posición en la que se encuentra actualmente.

Ejecutar hasta alcanzar una posición de memoria <Alt>+<F9>

Esta combinación de teclas es similar a F4. Aquí tiene la posibilidad de introducir directamente la posición en el programa (segmento y offset) hasta la que debe ser ejecutado el mismo (partiendo de la posición actual).

Mostrar pantalla de usuario<Alt>+<F5>

Al depurar debería disponer siempre de la posibilidad de volver a ver qué salidas se hicieron últimamente en la pantalla de usuario o durante qué entrada se detuvo el programa mediante un Breakpoint. Estas posibilidades se las ofrece el Turbo Debugger mediante las teclas Alt + F5. Si la aplicación se ha ejecutado en modo gráfico, se conmuta automáticamente a este modo. Sin embargo no se tiene en cuenta la reprogramación de los registros de la tarjeta VGA. Así que si su programa emplea el modo X de la tarjeta VGA, verá gráficos bastante confusos.

Buscar una expresión <Ctrl>+<S>

Si busca una expresión determinada en el programa a analizar, como por ejemplo el decremento de una variable, puede realizar la búsqueda mediante Ctrl + S. En la ventana de entrada que aparece introduzca la expresión en ensamblador que quiere encontrar, por ejemplo:

```
dec byte ptr [0054]
```

El Turbo Debugger busca desde la posición actual en el programa en adelante la aparición de esta combinación. Cuidado con instrucciones cortas. Aquí es posible que una parte de la instrucción se interprete como comando. En ese caso debería comprobar mediante un breve desplazamiento hacia arriba y abajo, si se trata realmente de la expresión buscada o de parte de otra.

Ir a <Ctrl>+<G>

El Turbo Debugger indica constantemente la posición en la RAM a la izquierda de la ventana de CPU. Si ha encontrado un lugar interesante en el programa, debería apuntar este lugar. Puede volver a él en cualquier momento mediante las teclas Ctrl + G. Esto simplifica la comprobación del programa en diferentes puntos.

Retorno al procedimiento invocador <Ctrl>+<C>

Si ha saltado a un procedimiento, para seguir el código, y resulta que en él no se encuentra lo que busca, no ha de atravesar todo el procedimiento hasta que llegue

al final. Con Ctrl + C puede volver a la llamada del procedimiento y saltarlo entonces con F8.

Ahora que está familiarizado con las funciones básicas del Turbo Debugger, debería estar en disposición de analizar cualquier programa. Por qué no intenta su suerte en el mini-juego RAIDER, que se encuentra como ejemplo de la programación de entrenadores en el CD adjunto. Este programa se puede depurar y modificar libremente según sus deseos.

¡Hexpléndido! - editores hexadecimales

El concepto "Hex-Editor" designa a un editor con el que no edita los archivos en el formato ASCII (letras) normal, sino que direcciona los caracteres con los códigos que se emplean en el sistema. Estos se reproducen en forma hexadecimal. De ahí el nombre del editor hexadecimal (Hex-Editor).

¿Para qué un editor hexadecimal?

¿Y para qué sirve un editor de este tipo? Hay archivos de los que no nos interesa el texto que se encuentra en ellos (como un ReadME.DOC), sino los números que almacenan. El mejor ejemplo para estos archivos son las partidas grabadas de un juego. Si le falta dinero en la simulación de bolsa o si en el simulador de vuelo tiene pocas armas, puede guardar el estado actual de la partida, y echarle un vistazo con un editor hexadecimal. Apúntese primero cuánto dinero tenía. Esta cantidad se ha de convertir ahora a hexadecimal. Digamos que tenía 1000 piezas de oro, que son *03E8h* piezas del oro. El procesador Intel sin embargo siempre guarda los números desde el byte pequeño al byte grande. Así que ha de invertir el orden. Ahora tiene *E803h*. Con el editor hexadecimal puede buscar justamente esta suma. Si la ha encontrado, sustitúyala por ejemplo por *9999h* e inicie el juego de nuevo. Si vuelve a cargar la partida, y ha encontrado el lugar adecuado del archivo, ahora podrá alegrarse de tener *39.321* piezas de oro.

Si no ha funcionado, debería seguir buscando, para ver si la cantidad aparece en otro lugar. De esta forma se pueden incrementar las vidas, municiones, armas extra, etc. Este sistema funciona para todos los archivos existentes en el ordenador. De esta forma puede buscar cualquier combinación de bytes y sustituirla por otra. Lo que naturalmente necesita es un buen editor.

El editor hexadecimal Hexcalibur

Uno de los mejores editores hexadecimales seguramente es Hexcalibur. El editor destaca porque no simplemente puede sobrescribir datos, sino también insertar-

los, eliminarlos y copiarlos. Para iniciar el editor ha de indicar el nombre del archivo a editar. Así que introduzca por ejemplo:

```
HC SAVEGAME.01
```

Pulse ahora una tecla cualquiera, y se encontrada en el editor. Este se ha colocado automáticamente en modo de inserción. Si quiere editar un Savegame, debería activar el modo de sobrescritura mediante una pulsación de la tecla [Ins]. En el editor se puede mover con las teclas del cursor. Puede pasar de la ventana hexadecimal de la izquierda y la ventana ASCII de la derecha con [F2]. Aquí tiene un resumen de las teclas definidas:

Teclas	Significado
Teclas de cursor	Mediante las teclas de cursor puede moverse en la ventana correspondiente.
[Av Pág]/[RePág]	Mediante estas dos teclas puede saltar una "página" hacia arriba o abajo. Las páginas miden exactamente 256 caracteres.
[Inicio]	Con esta tecla va al inicio de la línea actual.
[Fin]	Con esta tecla salta al final de la línea activa.
[Alt] + [B] o [Alt] + [0]	Mediante la pulsación de estas dos combinaciones de teclas puede llegar al principio del archivo que se encuentra en el editor:
[Alt] + [1] hasta [Alt] + [9]	Estas combinaciones de teclas sirven para un movimiento rápido en el archivo. Con [Alt] + [1] va al la posición que representa el 10% del archivo, con [Alt] + [2] llega al 20%, etc.
[Alt] + [E]	Mediante [Alt] + [E] salta al final del archivo.
[Alt] + [G]	Con esta combinación de tecla puede saltar a un punto determinado del disco. Para ello introduzca el sector y el número del primer byte y pulse a continuación [Return].
[F1]	Da un breve resumen de las funciones disponibles del editor.
[F2]	Con [F2] puede saltar de la ventana hexadecimal a la ventana ASCII y al revés.
[Ins]	Mediante [Ins] conmuta entre el modo de inserción y de sobrescritura.
[Supr]	Con esta tecla borra el carácter en el que se encuentra el cursor. El carácter se borra tanto en la ventana ASCII, como en la ventana hexadecimal. Atención: La pérdida de este carácter puede traer consigo errores fatales en archivos que pertenecen a programas.
[Alt] + [A]	Aquí puede seleccionar el principio del bloque con el que quiere realizar las siguientes operaciones.
[Alt] + [C]	Mediante [Alt] + [C] copia el bloque seleccionado.
[Alt] + [M]	Con esta combinación de teclas puede mover el bloque seleccionado.

Teclas	Significado
Alt + D	Para borrar un bloque pulse Alt + D. Atención: También aquí se aplica: ¡Borrado es borrado!
Alt + P	Mediante Alt + P puede elegir el bloque seleccionado para insertar.
Alt + F	Esta es una de las funciones más importantes del editor. A través de ella puede encontrar texto. Para ello simplemente introduzca el texto a buscar después de Alt + F, y se buscará desde la posición actual en adelante.
Alt + R	También esta función es muy útil. Puede sustituir una secuencia de caracteres cualquiera por otra. Las secuencias han de tener la misma longitud.
Alt + S	Para guardar el archivo (¿editado?) pulse esta combinación.
Alt + X, Alt + Q	Así abandona el programa. Si ha realizado modificaciones en el archivo, se le preguntará antes de salir si quiere guardar los cambios.
Alt + U	Para deshacer todos los cambios desde el último guardar, ha de pulsar Alt + U.

¿Cómo puede editar ahora una partida grabada con Hexcalibur? Supongamos que el archivo que quiere editar se llama SAVEGAME.001. Proceda de la siguiente forma: primero ha de iniciar al editor mediante HC SAVEGAME.001. La partida grabada se carga en el editor. Ahora busque el valor a modificar. Si por ejemplo tenía 1234 piezas de oro, pulse primero Alt + F, e introduzca a continuación *D2 04*, seguido de Return. HC busca ahora esta secuencia de bytes. Cuando la encuentre, selecciona la zona en rojo y termina la búsqueda. Ahora se encuentra en la posición que tiene que modificar. Introduzca el valor hexadecimal en la ventana correspondiente, por ejemplo *30 75* para *30.000*. Ahora pulse Alt + S para guardar los cambios. Si confirma la pregunta, se guarda la versión modificada. HC crea por defecto una copia de seguridad en forma de un archivo .BAK. Con ello se ha hecho el trabajo, y sólo le queda salir del programa mediante Alt + X. Si la primera aparición de la secuencia de bytes no hubiera sido el valor deseado, renombre la copia de seguridad del archivo con el nombre original, y busque la siguiente aparición de la secuencia. Esto lo puede hacer pulsando de nuevo Alt + F cuando HC encuentre el primer valor.

10.2 Las interrupciones de depuración

Para protegerse de la depuración de un programa naturalmente ha de saber cómo funciona un debugger. En el apartado 10.1 ya ha conocido extensamente el Turbo Debugger. ¿Pero cómo trabaja un depurador de forma genérica? Según qué calidad tiene el debugger, emplea la misma máquina virtual que el programa ejecuta-

do, o bien emula una CPU virtual. Todos los depuradores tienen en común que ocupan la interrupción *3*, la interrupción de depuración. Cuando se llama a esta interrupción, se activa un posible depurador que esté esperando en segundo plano. Si la interrupción no está instalada, el programa sigue funcionando.

Esto no lleva a la primera trampa que podemos montar en nuestros programa. Algunas llamadas a la *Int 3*, alegremente distribuidas por el programa y completamente aleatorias y sin relación, han "mosqueado" a más de un Hacker (junior).

Enmascarar interrupciones

Seguramente sería más efectivo enmascarar la *Int 3* por hardware. Esto funciona estupendamente para el teclado. Se encuentra en el controlador de interrupciones (port *20h*) en el bit 1 y por ello tiene la interrupción *09h*. Si simplemente bloqueamos el canal para esta interrupción, se puede llamar tantas veces como se quiera, la rutina correspondiente no se llamará. Sin embargo, la interrupción de depuración tiene una pequeña desventaja. Tiene un número menor de *8* y por ello es una NMI, una *Non maskable Interrupt*. El método de enmascarar la interrupción simplemente no funciona. Sin embargo, el enmascaramiento es un método sencillo para anular por ejemplo los puertos serie. Y sin ratón hay algunos programas que no funcionan...

La desviación de las interrupciones de debugging

Otra posibilidad de emplear las interrupciones es la de desviarlas a una rutina propia. Así por ejemplo puede descargar como procedimiento alguna rutina importante, que no llame demasiado, pero que sea imprescindible para la ejecución del programa y declararla como interrupción. Después desvía la interrupción *3* a su propia rutina y llama a la interrupción *3*. Sin embargo, este método sólo funciona para depuradores que no disponen de CPU virtual con tabla de interrupciones propia. Al Turbo Debugger no se le puede tomar el pelo tan fácilmente.

Guardar datos en el vector

Otro método muy popular es el guardar datos en el vector de interrupciones. El vector tiene espacio para un *DWORD*, y mientras no se llame, allí puede figurar en principio cualquier valor. A usted le queda la decisión se si simplemente escribe la suma de control en la posición de memoria para comprobarla en tiempo de ejecución, o si espera que se dispare la *Int 3* y el programa aterrice en algún lugar del Nirvana. La posición de la interrupción en la tabla de interrupciones se puede

encontrar en la posición de memoria *[0:4 * NÚMERO de interrupción]*. Normalmente se encontraría aquí la dirección de la rutina de manejo de la interrupción. Lo que escriba allí durante el tiempo de ejecución es asunto suyo. Pero en cualquier caso debería restaurar los vectores originales al abandonar su programa.

La posibilidad de poder acceder directamente a la tabla de vectores le ofrece otra opción como también la puede encontrar en muchos productos comerciales. Los vectores de las interrupciones necesitadas por el programa, como por ejemplo la interrupción de la Sound Blaster, no se desvían mediante las funciones *25h* y *35h* de la *Int 21h*, sino mediante un acceso directo a la tabla de vectores. Sin embargo hemos de decir que posibles trucos con los vectores sólo protegen contra el debugging, si el depurador no emplea ninguna máquina virtual.

Los siguientes procedimientos de ensamblador muestran el empleo de este método Los puede encontrar en NODEB.ASM. Con el procedimiento *Check_if_vector* guarda el valor *Longint* pasado en la tabla *Vector* en la posición de la *Int 3*,

```
public Check_if_vector
Check_if_vector proc pascal check : dword;
  mov bx, 0
  mov es, bx
  mov bx, 18
  mov eax, es:[bx]
  mov oldint3, eax
  mov eax, check
  mov es:[bx],eax
  ret
Check_if_vector endp
```

Mediante *Vector_ok* puede comprobar ahora si hay guardado un valor *Longint* determinado en la tabla de vectores en la posición de la *Int 3*. Se devuelve *1* o TRUE si el valor de la tabla coincide con el valor pasado, sino obtiene *0* o *FALSE*.

```
public Vector_ok
Vector_ok proc pascal check : dword;
  mov bx, 0
  mov es, bx
  mov bx, 18
  mov eax, es:[bx]
  cmp eax, check
  je  @check_ok
  mov al, 0
  jmp @check_fin
@check_ok:
  mov al, 1
```

```
@check_fin:
  ret
Vector_ok endp
```

El procedimiento *Restore_Checkvector* sirve para volver a su estado original a la *Int 3* desviada. Este procedimiento se debería ejecutar en cualquier caso, si ha desviado el vector.

```
public restore_Checkvector
restore_Checkvector proc pascal
  mov bx, 0
  mov es, bx
  mov bx, 18
  mov eax, oldint3
  mov es:[bx],eax
  ret
restore_Checkvector endp
```

Sustituir interrupciones

La última trampa en relación a las interrupciones es la sustitución de una interrupción por otra. En la práctica esto significa que ya no llama a la interrupción *21h*, sino a la interrupción paso a paso (*01h*) o la interrupción de depuración (*03h*). Este método es muy útil cuando programa en ensamblador (al menos los procedimientos de clave de acceso se deberían realizar en el Kernel y en ensamblador). Obtenga primero la dirección de la *Int 21h*, y guárdela en un vector cualquiera. Si ahora quiere llamar la *Int 21h*, escriba en vez del *21h* el número del vector ocupado por usted. Este método no funciona obligatoriamente, pero en cualquier caso no queda bonito, ya que reduce mucho la legibilidad de código. Para usted no deberían aparecer problemas en la programación, ya que puede realizar la sustitución de las interrupciones cuando el programa esté acabado.

Puede encontrar un ejemplo de como puede copiar la interrupción *21h* sobre la interrupción *3h* en el procedimiento de ensamblador *Copy_int21_int3*. La interrupción antigua se guarda en la variable *antigua_interrupt3* y debería restaurarse antes de finalizar el programa

```
public Copy_int21_int3
Copy_int21_int3 proc pascal
  mov bx, 0
  mov es, bx
  mov bx, 18
  mov eax, es:[bx]
  mov antigua_interrupt3, eax        ; guardar interrupción 3 antigua
  mov bx, 84                         ; cargar interrupción 21
  mov eax, es:[bx]
```

```
  mov bx, 18                          ; guardar en interrupción 3
  mov es:[bx],eax
  ret
Copy_int21_int3 endp
```

10.3 Engañar al debugger

Otra posibilidad es la de no hacer trucos generales en el programa, sino apagar claramente el depurador. Primero nos orientaremos en los debuggers generales, pero recordando siempre al Turbo Debugger como referencia.

El truco de la memoria

El método más sencillo lo representa el truco de la memoria. No reserva la memoria necesitada para su programa, sino toda la disponible. En Turbo Pascal lo puede hacer mediante la instrucción

```
{$M $800,550000,655000}
```

Si ahora intenta depurar el programa con el *Turbo Debugger* normal, obtendrá el mensaje de error de que no hay memoria suficiente para la ejecución del programa. Si coloca el valor lo suficientemente alto, incluso puede conseguir que el *Turbo Debugger386* tire la toalla. Pero ha de tener en cuenta de no sobretensar el arco, ya que no todos los ordenadores están configurados de modo que tengan 630 KBytes de memoria disponibles. Y en esos ordenadores su programa deberá funcionar, ¿no?

Instrucciones de doble sentido

Otra posibilidad de tomarle el pelo directamente al debugger, es el empleo de instrucciones de doble sentido. Con ello nos referimos por ejemplo al salto en medio de un comando. La mayoría de depuradores no se aclaran con esta instrucción y saltan a la siguiente reconocible para ellos. El truco consiste en escribir en la siguiente instrucción un comando para terminar el programa. Si el programa es depurado paso a paso, el depurador se va a la playa en este lugar. En funcionamiento normal sin embargo, salta a la instrucción y allí encuentra otra instrucción de salto que le lleva más allá de la instrucción de *Stop*. Aquí tiene un ejemplo de como puede ser todo esto en la práctica (*ds = cs*):

```
No_Stepping proc near
  push ax
  jmp @Nostep + 2
```

```
@Nostep:
  mov ds:byte ptr [06EBh],00
  mov ax, 4C01h
  int 21h
  pop ax
  ret
No_Stepping endp
```

Detener el TD386

Los métodos presentados hasta ahora se han orientado todos preferentemente en el TD normal. Pero precisamente en los métodos orientados al hardware el *TD386* nos tiene ventaja. Esta, naturalmente, es una circunstancia que no vamos a permitir. Ya también para él hay un método seguro: el *TD386* lo aguanta todo, ¡excepto el Protected Mode! A primera vista esto puede parecer decepcionante, pero le podemos tranquilizar. Todo el truco sólo consta de una pocas líneas. No nos hemos de preocupar de todo el aparato de gestión del Protected Mode. Basta con conmutar al Protected Mode y volver inmediatamente. Esto se puede realizar mediante el registro *cr0*. Todo esto ya no funcionará en ordenadores inferiores a *286*, pero esta familia de máquinas está desapareciendo rápidamente del mundo de la informática. Aquí tiene el procedimiento que conmuta al Protected Mode e inmediatamente después vuelve al modo real. Por cierto, también funciona con un controlador EMS instalado.

```
public protected_stopping
protected_stopping proc pascal
   pusha
   cli                          ; Desactivar interrupciones
   mov eax, cr0                 ; Conmutar al Protected-Mode
   or eax, 1
   mov cr0, eax
   jmp PROTECTION_ENABLED       ; Borrar Executionpipe
PROTECTION_ENABLED:
   and al, 0FEh                 ; Volver al modo real
   mov cr0, eax                 ; No reiniciar la CPU
   jmp PROTECTION_DISABLED      ; Borrar Executionpipe
PROTECTION_DISABLED:
   sti                          ; Reactivar interrupciones
   popa
  ret
protected_stopping endp
```

10.4 Automodificaciones

Una posibilidad muy interesante y muy extendida de proteger programas es la automodificación. Esto significa que el PC ejecuta el programa de forma totalmente distinta a la que se carga en memoria. En este apartado se incluyen las sumas de control y varios métodos de encriptar los datos de forma general.

Empleo correcto de las sumas de control

Si es el orgulloso propietario de un Modem, seguramente conocerá las sumas de control. Sirven, por ejemplo, para comprobar si un archivo tiene el tamaño correcto o si se encuentran los datos correctos en él. Para la transmisión de datos vía Modem esto es necesario comprobar todo el archivo transferido. En los programas esto también se podría hacer, pero no tiene necesariamente sentido. Al fin y al cabo, no todo el código del programa es interesante. Lo mejor es concentrarse en los tramos que hay alrededor de su control de clave de acceso. Esto también se aplica a posibles claves depositadas en el programa. Estos lugares no se pueden localizar así como así, pero se puede emplear un pequeño truco: al principio del procedimiento declara una constante local con una secuencia característica. Si tomamos como ejemplo la constante *Longint 12345678h*, esta se puede encontrar con facilidad en el compilado acabado. Simplemente anote su posición en el archivo EXE, y obtenga la suma de control de los siguientes 100 a 500 bytes. Con ello puede asegurar que nadie ha tocado su programa.

El mismo método también funciona con el programa en memoria. Es fácil obtener la dirección del procedimiento más interesante. Compruebe, partiendo de esta dirección, los siguientes 100 a 500 bytes y puede estar seguro de que el procedimiento no ha sido modificado por un TSR-Crack. Su procedimiento de suma de control debería estar asegurado una o varias veces por otros procedimientos de suma de control. Debe llamar a estos procedimientos de forma aleatoria en su programa. Especialmente ingenioso es si sólo realiza las comprobaciones después de que haya pasado un cierto tiempo, o bien cuando el usuario quiera guardar su trabajo a la partida...

Algoritmos de encriptación

Los algoritmos de encriptación son un campo muy amplio, con el que seguramente se podría llenar un libro propio. Aquí sólo describiremos brevemente los principios básicos. Lo primero es modificar el carácter en sí con una clave (key) determinada, que suele ser otro carácter. Un método sencillo pero efecti-

vo es el de sumarle la clave al carácter. Por suerte el procesador posee un tratamiento de rebase (overflow), y comienza a contar de nuevo desde 0 después de llegar al valor máximo. Esto significa que no se pierden datos, aunque se trabaje con una clave muy alta.

En principio la operación con un *XOR* es lo mismo. La codificación se realiza mediante operación con la clave, y para decodificar se ha de realizar exactamente la misma codificación. Este método es efectivo cuando se emplea junto con un número aleatorio que no se guarda. La gracia de este asunto está en que los números aleatorios no son realmente tan aleatorios. Si trabaja con la misma inicialización, siempre obtendrá la misma secuencia de números. Por ello los podrá emplear sin problemas

Otra posibilidad de encriptación es el empleo de un alfabeto propio. En este método, todos los caracteres se sustituyen por un carácter de su propio alfabeto. Esta forma es bastante más difícil de desencriptar que la forma presentada anteriormente, ya que no sólo se necesita la clave, sino todo el alfabeto completo. Como contrapartida la decodificación tarda más tiempo. Aunque si sólo la aplica sobre determinadas zonas de un archivo, puede ser muy apropiada.

El truco PIQ

El truco PIQ ahora no sólo es teoría gris, sino un ejemplo contundente de cómo puede detener realmente a cualquier debugger. El truco se basa en la Prefetch-Queue de la CPU. La CPU no sólo lee la instrucción que va a procesar, sino también la siguiente, para poder trabajar más deprisa.

El truco consiste en modificar una posición de memoria adyacente. Así por ejemplo puede convertir una simple comprobación de la versión del DOS en una instrucción para detener el programa. Si éste se ejecuta con normalidad, la instrucción no provoca más problemas, ya que ya ha sido cargada por la PIQ. La posición de memoria se modifica, pero esto ya sólo nos interesa, si queremos volver a acceder a esta posición. Pero hay activo un debugger, que realiza las instrucciones paso a paso, la PIQ no está activa. Esto significa que la instrucción ya se ejecuta la primera vez. Y con ello se detiene el programa. Aquí tiene el código de cómo se puede aplicar este truco.

```
PIQ_Stop_System proc near
    push ds
    push ax
    push bx
    push cs
```

```
  pop  ds                          ;CS a DS
  mov  cs:word ptr [@int_21_funkt],4CB4h  ; Función terminar programa
@int_21_funkt:
  mov  ah, 30h   ; Función averiguar versión del DOS
  int  21h
  pop  bx
  pop  ax
  pop  ds
  ret
PIQ_Stop_System endp
```

Por favor tenga en cuenta que por razones de mejor visibilidad hemos representado todos los trucos en procedimientos individuales. En sus programas naturalmente no deberá emplear estos procedimientos, sino implementar directamente el código. De lo contrario, la mejor trampa se puede eliminar de su programa en un abrir y cerrar de ojos, sobrescribiendo simplemente la llamada del procedimiento con *nops* (*non operate, 90h*).

El empleo de programas compresores

Es muy aconsejable el empleo de programa compresores. Tienen dos efectos secundarios muy ventajosos: por una parte el archivo EXE ya no ocupa tanto espacio en el disco duro, por la otra ofrecen una estupenda encriptación. Como el archivo EXE completo se presenta de forma comprimida, ya no es posible parchearlo (realizar un patch). A pesar de que el programa se puede descomprimir con los programas presentados en el capítulo 10.1, pero cuando se comprime de nuevo el archivo, suele quedarse en un tamaño diferente que el original. Si en el programa se ha previsto un control del tamaño del mismo, se puede interrumpir la ejecución con un mensaje de error.

Los métodos aquí presentados frecuentemente no son más que una molesta barrera, que se puede eliminar con cierta velocidad. Pero combinados de la forma adecuada, se convierten en una protección bastante eficiente. Es muy aconsejable el colocar las consultas en bucles lo más grande posibles. Controle estos bucles por una multitud de variables globales, esto hace que el código sea muy difícil de comprender. Las llamadas diseminadas a la *int 3h* también son muy aconsejables. Y acuérdese de no emplear los trucos en procedimientos, sino directamente en el código. Los métodos expuestos se emplean de esta forma en muchos productos comerciales. Con ello debería estar en disposición de complicarle la vida a un buen número de Hackers.

10.5 Entrénese - entrenadores y depuradores de juegos

Los entrenadores son uno de los campos más fascinantes de los ordenadores. Bajo un entrenador se entiende un programa que suele ser un *TSR* o un cargador, que habitualmente pone a disposición del usuario algunas funciones mediante una tecla abreviada, con las que se puede intervenir en otro programa. Este método es utilizado, lamentablemente, por algunos Crackers para saltarse las protecciones. Pero aquí no nos ocuparemos de estas cosas.

Echaremos un vistazo al género de los entrenadores de juegos. ¿Quién no lo conoce? El estupendo juego de acción, en el uno se pelea hasta el final con todos, y cuando se llega al jefe sólo se tiene una vida, y casi ningún arma. En ese caso se puede estar contento de disponer de un entrenador. Simplemente se pulsa una combinación de teclas y ya se vuelve a disponer de todas las armas y todas las vidas.

El problema es que estos entrenadores no existen para todos los juegos. Así que sólo se puede hacer una cosa: escribir uno. En este capítulo puede aprender todo lo que ha de saber, para poder realizar un entrenador. Primero hemos de pensar qué programas son aptos para entrenar y cuales no lo son tanto. Los juegos de rol y las aventuras de Sierra™ son completamente inadecuados para estos propósitos. Este género de juegos emplean una especie de lenguaje descriptivo que es traducido por un intérprete. Aquí puede ser interesante guardar la partida y editarla con un editor hexadecimal. En un caso extremo puede escribir un editor de Savegame.

Todos los tipos de juegos de acción son muy adecuados para entrenarse. Aquí se descuentan vidas, se reparten armas extra y se recogen puntos. Todo ello se gestiona en el juego en forma de variables *WORD* o *DWORD*. Para un entrenador no ha de hacer nada más que encontrar la posición de estas variables en la memoria y escribir un pequeño TSR que llene con valores nuevos estas variables, al pulsar una tecla determinada.

¡Busque! - encontrar variables

Bueno, si fuera tan sencillo como suena, para cada juego habría una docena de entrenadores. Pero no es tan fácil. El mayor problema lo representa la localización de las variables empleadas. Para ello necesita un depurador y el Turbo Debugger sería bastante adecuado para la tarea. Cómo lo puede emplear con

efectividad lo puede aprender en el apartado 10.1. Primero debería buscar las estructuras comunes en las que se incrementan o decrementan variables. Supongamos que busca el lugar del programa en el que se pierden las vidas. Si encuentra este lugar, ya ha encontrado la variable de las vidas. Para encontrarla, hay dos métodos:

Método A - Búsqueda de estructuras empleadas comúnmente

Primero debería probar este método, ya que, si tiene éxito, lleva antes hasta el objetivo. En nuestro ejemplo buscamos una secuencia de bytes que decrementa el contenido de una variable de bytes. Es una variable byte, porque suele ser imposible que obtenga más de 256 vidas. La estructura empleada con más frecuencia para decrementar es:

```
dec byte ptr [xxyy]
```

donde *xxyy* es la posición offset de la variable en el segmento de datos actual. El ensamblador traduce esta instrucción en

```
FE 0E yy xx.
```

Esto significa para usted en la práctica, que ha de buscar la combinación de bytes *FE 0E*, ya que no conoce *yy xx*. No se sorprenda en programas grandes de cuántas combinaciones encontrará. Para comenzar, habrá toda una serie de combinaciones que no procedan, ya que son parte de una instrucción de más bytes. Luego, mucho *Decrements* se referirán a la misma variable. Apunte las variables encontradas, y compruebe su valor durante la ejecución del programa. Y ya ha encontrado lo que buscaba.

Si no consigue nada a través del comando *dec*, debería buscar otra combinación corriente. La siguiente tabla le ofrecerá un resumen de las estructuras más comunes:

Instrucción ensamblador	Traducción
dec word ptr [xxyy]	FF 0E yy xx
dec byte ptr [xxyy]	FE 0E yy xx
sub word ptr [xxyy],zz	83 2E yy xx zzzz
sub byte ptr [xxyy],zz	80 2E yy xx zz
sub [xxyy],ax	29 06 yy xx
sub [xxyy],dx	29 16 yy xx
inc word ptr [xxyy]	FF 06 yy xx
inc byte ptr [xxyy]	FE 06 yy xx
add word ptr [xxyy],zz	83 06 yy xx zzzz
add byte ptr [xxyy],zz	80 06 yy xx zz

Tenga en cuenta durante su búsqueda, que algunos programas emplean varios segmentos de datos, o que guardan datos en el segmento de código. Así que si no encuentra nada en el segmento actual, debería buscar en los demás que emplea el programa. Si después de analizar todos los segmentos no obtiene el valor buscado, o simplemente obtiene demasiados valores adecuados, tendrá que tomar el duro camino B.

Método B - Seguir la secuencia del programa

El trazado de programas ha obtenido algo de esplendor y peligro desde el juicio entre Microsoft y Stack Inc. Pero si el método A fallara, es la única posibilidad de averiguar las variables buscadas. Para ello ha de emplear la siguiente estrategia: primero ha de localizar un procedimiento en la que se reconstruya la pantalla. Este procedimiento se ha de repasar paso a paso, viendo que variables son empleadas para dibujar la pantalla. Aquí podrá realizar progresos con bastante celeridad encontrando fácilmente una serie de variables adecuadas. Si no encontrara la variable para las vidas o el final del programa, siga al procedimiento principal hasta que llegue a la comparación de una variable con 0. Este no necesariamente es la variable para las vidas o el final del programa, aunque suele serlo.

Manejo de los valores delimitadores

Después de haber encontrado las variables empleadas por el programa puede comenzar escribiendo el entrenador. Para ello necesita los valores delimitadores del programa. Primero ha de apuntar el segmento de datos de las diferentes variables. Después ha de averiguar si el programa emplea una rutina propia para el manejo del teclado. Busque la instrucción

```
in al, 60h
```

en el programa. Coloque un Breakpoint en el programa en ese punto. Ejecute el programa, y pulse allí una tecla. Si el programa se interrumpe y se encuentra en la posición del Breakpoint, ha encontrado la rutina de manejo. Apunte el segmento de código actual (CS) y el offset en el que se encuentra la instrucción. Este valor se puede encontrar en el Instruction Pointer (IP). Si el programa no se interrumpe, ha de seguir buscando la instrucción. Si no lo encuentra, compruebe si su programa dispone de varios segmentos de código. Ha de buscar en todos ellos, si no lo encuentra en el segmento base. Frecuentemente se recomienda echarle un vistazo a los segmentos *ES* y *DS*.

Si a pesar de todos los esfuerzos no encuentra nada, es muy probable que su programa no disponga de rutina de manejo independiente, sino que emplee la interrupción *9h* o *16h* del DOS.

Así se programan entrenadores

¿Cuál es la estructura de un entrenador? Los entrenadores habitualmente se componen de dos partes fundamentales: por una parte la rutina de instalación. Esto no significa la carga del programa como TSR, sino la integración en el programa a entrenar. La segunda parte es la rutina de interfaz. En ella se comprueba si se ha pulsado una tecla del entrenador, para reaccionar de la forma adecuada.

La parte de instalación

Para la instalación ha de diferenciar si el programa emplea sus propias rutinas de teclado, o si accede a las del DOS. En caso de empleo de la rutinas del DOS, simplemente ha de desviar la interrupción *09h* a su propio programa. Allí puede comprobar si se ha pulsado alguna tecla del entrenador, reaccionar en consecuencia y llamar a la rutina original después de ello.

Si el programa emplea su rutina propia, la cosa se complica. Para este propósito primero necesitamos una interrupción libre. Para ello podemos emplear la interrupción *65h*. Después nos "enganchamos" a una interrupción llamada frecuentemente, como por ejemplo la interrupción *21h*. Esta será llamada con toda seguridad por el programa a entrenar. Mediante ella sólo ha de realizar una pequeña modificación del código del programa. Esto es legal, ya que no modifica el programa en el disco duro, sino en la memoria. El lugar a modificar es el punto en el programa en el que se lee un carácter del teclado mediante

```
in al, 60h.
```

En Opcodes la instrucción es *E4 60*. Si ahora se genera una interrupción, el procesador guarda las banderas, el segmento de código desde el que se realizó la llamada y el Instruction Pointer actual en el stack. Guarde ahora los valores actuales de *BP*, *BX* y DS en el stack. Mueva el registro de la pila desde *SP* a *BP*. Ahora puede obtener el segmento de código mediante *[BP + 10]* y la posición del *IP* mediante *[BP + 8]*. Mueva el valor del segmento de código a *DS* y el *IP* a *BX*. Ahora puede comprobar si en la dirección de memoria direccionada mediante *DS:[BX]* se encuentra *E4 60*. Si es así, la interrupción fue dispara por el programa a entrenar y ha encontrado el lugar a modificar. Sustituya los dos bytes por *CD 65*. En vez de leer un carácter mediante *in al, 60h*, se llama la interrupción 65h.

En esta interrupción *65h* no debe modificar ningún registro excepto el *al*. Además ha de leer en cualquier caso un carácter del teclado mediante *in al, 60h*. Compruebe si el carácter en una tecla de su entrenado. Si es así, se procesa el código correspondiente del entrenador, de lo contrario se termina la interrupción.

La parte de interfaz

La parte de interfaz es la parte realmente interesante del entrenador. También aquí tendrá que volver a realizar dos diferenciaciones. Concretamente entre programas que sólo emplean un segmento de datos y aquellos que emplean varios.

Dediquémonos a los de la primera especie. Habitualmente se trata de programas que fueron creados por un lenguaje de programación como por ejemplo Borland Pascal y que sólo permiten un segmento de datos. Aquí no se ha de preocupar más de si emplea el segmento correcto, ya que esto suele ser así automáticamente. Si se llama la interrupción desviada por usted, se ha de comprobar primero qué tecla se ha pulsado. Si es una tecla del entrenador, la rutina primero salta a la parte en la que se modifica la variable correspondiente y finalmente a la sección en la que la interrupción termina correctamente. Esto suele ser, según la interrupción que emplee, un *IRET* o la llamada de la interrupción original. Si la tecla no es ninguna del entrenador, esta parte se accede directamente.

En la parte que modifica las variables, escriba ahora el nuevo valor en la posición que ha averiguado para la variable. No siempre tiene sentido el tomar el valor máximo para el tipo de variable. Debería averiguar en qué valor la energía está completamente restaurada o se dispone del máximo de armas. Una instrucción típica podría tener el siguiente aspecto:

```
Full_Energy:
mov ds:[1234],80
jmp Terminar_rutina
```

La cosa se complica si dispone de un programa con varios segmentos de datos. El segmento correspondiente de la variable a modificar ya lo debería tener apuntado. También conoce el segmento de código en el que se consulta el teclado. De este puede inferir el segmento de datos. En la interrupción del entrenador debe obtener el segmento de código durante la llamada de la interrupción. A este valor hay que sumarle o restarle la diferencia entre el segmento de datos de la variable y el segmento de código durante la llamada de la interrupción. Ahora ya dispone del segmento de datos correcto. Ya puede modificar la variable con toda normalidad. Aquí tiene un ejemplo de cómo se podría realizar la rutina:

```
Full_weapons:
  pusha
  mov bx,[bp + 8]
  sub bx, 3E43h
  mov ds, bx
  mov ds:[1234],255
  popa
  jmp Terminar_interrupción
```

Y estos ya han sido todos los secretos que ha de saber para poder escribir un entrenador. A lo mejor ahora puede intentar escribir un entrenador para el mini-juego del libro, RAIDER.

Entrenador para el juego RAIDER

Esperemos que usted haya intentado su habilidad en el juego RAIDER. NO es muy difícil de entrenar, y sirve como buen punto de entrada.

Si este ha sido su primer entrenador, seguramente aún ha tenido que luchar con algunas dificultades. En el CD adjunto puede encontrar el código completo del entrenador del juego. El código está pensado como kit base para sus propias aplicaciones. Aquí hablaremos más extensamente de la parte de interrupciones, que contiene realmente la auténtica programación del entrenador. Primero el entrenador define algunas variables y un identificador que sirve para saber si el entrenador ya está residente.

```
;               ####################################
;               ###                              ###
;               ###   Trainer de R A I D E R      ###
;               ###                              ###
;               ###   (c) 1994 by DATA Becker    ###
;               ###                              ###
;               ###   Autor: Boris Bertelsons    ###
;               ###                              ###
;               ####################################
;
;

.286
  w equ word ptr
  b equ byte ptr
code segment public
public insthand
public handler9
public reslim
public oldint9
```

```
public Identificador
public klen
public check_inst

assume cs:code,ds:code
Identificador:  db 'Trainer para DATA BECKER by InspirE /TC'
oldint9: dd 0
procedimiento: dd ?
klen equ offset oldint9 - offset Identificador

;#####################################################################
;###                                                               ###
;### Aquí se encuentran las verdaderas rutinas del trainer         ###
;###                                                               ###
;#####################################################################
```

A continuación se instala el nuevo controlador para la interrupción *9h*. Este lee un carácter del teclado y comprueba si es un carácter del entrenador. Si es así, se adapta la variable correspondiente. Las variables se obtienen mediante una pequeña sesión del Turbo Debugger.

```
;#####################################################################
;###                                                               ###
;### La nueva INT 9h. El procedimiento verifica si se pulsa una    ###
;### tecla del entrenador y modifica las variables.                 ###
;###                                                               ###
;#####################################################################

handler9 proc pascal
  pushf
  push bp
  push ds
  push bx

  in al, 60h        ; leer carácter

  cmp al,59         ; Tecla F1
  je Game_over
  cmp al,63         ; Tecla F5
  je Full_level
  cmp al,64         ; Tecla F7
  je Add_Points
  jmp Fin_keyb

Final_normal:
Fin_keyb:
  pop  bx
```

```
    pop  ds
    pop  bp
    popf
    jmp dword ptr ds:oldint9      ;llamar anterior int 9h

  Game_over:
    pushf
    PUSHA
    ;¡Haga sus modificaciones!

    mov ds:byte ptr [0648h],1

    POPA
    popf
    jmp final_normal


  Full_level:
    pushf
    PUSHA
    ;¡Haga sus modificaciones!

    mov ds:byte ptr [0648h],255

    POPA
    popf
    jmp final_normal

  Add_points:
    pushf
    PUSHA
    ;¡Haga sus modificaciones!

    mov bx,ds:word ptr [064Ah]
    add bx,100d
    mov ds:word ptr [064Ah],bx
    POPA
    popf
    jmp final_normal

  handler9 endp
```

El código que sigue a continuación sirve para la instalación o desinstalación del entrenador:

```
  insthand proc pascal
  reslim label byte
    push ds
```

```
  pop ds
     mov ax,3509h                    ; guardar INT 21 antigua
  int 21h
  mov w oldint9,bx
  mov w oldint9 + 2,es
  mov ax,2509h                    ; INT 21h a propia rutina
  lea dx,handler9
  int 21h
        ret
insthand endp


check_inst proc near
  mov ax,3509h                    ; obtener vector de interr.
  int 21h
  mov di,bx
  mov si,offset Identificador
  mov di,si
  mov cx,klen
  repe cmpsb                      ; comprobar identificación
        ret
check_inst endp

code ends
end
```

El programa RAIDER sólo tiene un segmento de datos y no emplea ninguna rutina propia de teclado. Por ello este entrenador es muy adecuado para comenzar.

El siguiente entrenador representa un pequeño "bocati di cardinale", ya que además llama al procedimiento para la actualización de la representación de la variable modificada. También puede ver muy bien, cómo se maneja el control de los programas que programan el teclado directamente.

```
;               ####################################**
;               ###                                ###
;               ###   Trainer de #########         ###
;               ###                                ###
;               ###   (c) 1994 by DATA Becker      ###
;               ###                                ###
;               ###   Autor: Boris Bertelsons      ###
;               ###                                ###
;               ####################################**
;
;

.286
  w equ word ptr
  b equ byte ptr
```

```
code segment public
public insthand
public handler21
public reslim
public oldint21
public oldint65
public Identificador
public klen
public check_inst

assume cs:code,ds:code
Identificador:  db 'DATA BECKER'
oldint21: dd 0
oldint65: dd 0
procedimiento: dd ?
klen equ offset oldint21 - offset Identificador


;#######################################################################
;###                                                                 ###
;###En este lugar se encuentran las verdaderas rutinas del trainer###
;###                                                                 ###
;#######################################################################
```

Primero la nueva interrupción para la interrupción *21h*. Comprueba si en el programa se encuentra la instrucción *in al, 60h* y la sustituye, si es así:

```
;#######################################################################
;###                                                                 ###
;###  La nueva INT 21h. El procedimiento comprueba si en el lugar ###
;###  indicado en memoria se encuentra la instrucción «in al,60h» ###
;###  y lo sustituye por «int 65h» si procede!                       ###
;###                                                                 ###
;#######################################################################
handler21 proc pascal
  pushf
  push bp
  push ds
  push bx
  mov  bp,sp
  mov  bx,[bp+10]  ; cs durante la interr. a BX, DOS !!!
                   ; ATENCION! En TD [bp+16] !!!
  add  bx,0366h    ; CS del 1. INT 21h+2136h=CS de rutina de teclado
  mov  ds,bx       ; cs de la rutina de teclado a ds
  mov  bx,568Bh    ; 8B56h = mov dx,[bp+06]
  cmp  ds:word ptr [0005h],bx     ; está en la rutina de teclado ?
  jne  no_cambiar
  mov  ds:word ptr [0005h],9090h  ; escribir Int 65h!
  mov  ds:word ptr [0007h],65CDh  ; escribir Int 65h!
```

```
no_cambiar:
  pop  bx
  pop  ds
  pop  bp
  popf
  jmp dword ptr cs:oldint21        ; llamar Int 21 antigua
handler21 endp
```

Aquí tiene las verdaderas rutinas del entrenador. El controlador obtiene primero el segmento de código durante el tiempo de llamada de la interrupción. Ya que el segmento de código del manejo del teclado es el mismo que el de los procedimientos para la actualización de la pantalla, se puede adoptar directamente. De lo contrario se ha de realizar una adaptación según el método anteriormente descrito. Si se ha de entrenar una variable, primero se coloca el puntero al procedimiento con la actualización de pantalla correspondiente. A continuación el procedimiento salta al controlador original.

```
;#################################################################
;##                                                             ##
;##El proc. Int65h. Leer un caracter de «in al,60h» y combrueba si##
;##el caracter leido fue definido como tecla Trainer. Si es así  ##
;##se realizan las modificaciones de memoria asignadas y las     ##
;##llamadas de procedimientos. En este lugar debe incluir sus    ##
;##variables de entrenamiento!!!                                 ##
;##                                                             ##
;#################################################################

handler65 proc far
  pushf
  push bp
  push ds
  push bx
  mov  bp,sp
  mov  bx,[bp+10]  ; cs durante la interr. a BX
  in   al,60h      ; leer caracter
  cmp al,63        ; Tecla F5
  je  Full_Shoots_j
  cmp al,64        ; Tecla F6
  je  Full_Lives_J
  cmp al,65        ; Tecla F7
  je Weapon_new_j  ;
  cmp al,66        ; Tecla F6
  je Weapon_new_j  ;
  cmp al,67        ; Tecla F9
  je Weapon_new_j  ;
  cmp al,68        ; Tecla F10
  je  More_Points_J
```

```
Fin_Keyb:
  pop bx
  pop ds
  pop bp
  popf
  iret

Full_Shoots_j:
  jmp Full_Shoots
Full_Lives_j:
  jmp Full_Lives
More_Points_j:
  jmp More_Points
Weapon_new_j:
  jmp Weapon_new

Full_Shoots:
  pushf
  PUSHA
  sub  bx,0     ; ya que CS es ok
  mov word ptr procedimiento+2,bx
  mov  bx,1401h     ; es:[bx]  =  14EF:1401
  mov word ptr procedimiento,bx
;———
  mov ds:byte ptr [0DA3h],20h
  mov ax,20h
  push ax
  call dword ptr [procedimiento]
  POPA
  popf
  jmp Fin_Keyb

Full_Lives:
  pushf
  pusha
  sub  bx,0  ;
  mov word ptr procedimiento+2,bx
  mov  bx,1317h  ; es:[bx]  =  14EF:1317
  mov word ptr procedimiento,bx
;———
  mov ds:byte ptr [0DA3h],0009
  mov ax,9
  push ax
  call dword ptr [procedimiento]
  popa
  popf
  jmp Fin_Keyb

Weapon_new:
```

```
  pushf
  pusha
  sub  bx,0  ;
  mov word ptr procedimiento+2,bx
  mov  bx,1454h  ; es:[bx]  =  14EF:1454
  mov word ptr procedimiento,bx
;———
  sub al,65
  mov ah,0
  mov ds:byte ptr [0DA2h],al
  push ax
  call dword ptr [procedimiento]
  popa
  popf
  jmp Fin_Keyb

More_Points:
  pushf
  pusha
  sub  bx,0  ;
  mov word ptr procedimiento+2,bx
  mov  bx,1BD0h  ; es:[bx]  =  14EF:1BD0
  mov word ptr procedimiento,bx
;———
  mov ax,1000
  push ax
  call dword ptr [procedimiento]
  popa
  popf
  jmp Fin_Keyb

handler65 endp

insthand proc pascal
reslim label byte
  push ds
  pop ds
   mov ax,3521h                ; guardar INT 21 antigua
  int 21h
  mov w oldint21,bx
  mov w oldint21 + 2,es
   mov ax,3565h                ; guardar INT 65h antigua
  int 21h
  mov w oldint65,bx
  mov w oldint65 + 2,es
  mov ax,2521h                 ; INT 21h a propia rutina
  lea dx,handler21
  int 21h
  mov ax,2565h                 ; INT 65h propia rutina de teclado
```

```
  lea dx,handler65
  int 21h
   ret
insthand endp

check_inst proc near
  mov ax,3521h                    ; obtener vector de interr.
  int 21h
  mov di,bx
  mov si,offset Identificador
  mov di,si
  mov cx,klen
  repe cmpsb                      ; comprobar identificación
   ret
check_inst endp

code ends
end
```

Con ayuda de estos dos ejemplos debería estar en disposición de escribir un entrenador para la mayoría de juegos entrenables. Pero también aquí se aplica eso de: practicar, practicar y practicar...

11. Administración de la memoria

Todo PC moderno está equipado ya con al menos 4 MBytes de memoria principal, habitualmente con un tiempo de acceso de 70 ns o mejor. Bajo DOS sólo se pueden emplear 640 KBytes de esta memoria de forma directa. Cómo puede emplear el resto de la misma, lo aprenderá a lo largo de este capítulo. Comencemos con las estructuras fundamentales.

11.1 La memoria DOS convencional

La memoria convencional libre bajo DOS suele ser habitualmente de 640 KBytes. Muchas veces esto no suele ser suficiente para algún programa. Para eliminar esta falta, podrá aprender en este capítulo cómo puede aprovechar el modelo de memoria FLAT de los 80286, 80386, 80486 y Pentium, de modo que podrá gestionar hasta 4 GBytes de memoria. Esto debería ser suficiente para la mayoría de los programas.

La memoria convencional del DOS es, como ya hemos mencionado, de 640 KBytes. Sólo que no se puede direccionar como un gran bloque, sino que está dividido en los llamados segmentos. Un segmento siempre tiene 64 KBytes y comienza en una dirección de párrafo determinada. Una dirección de párrafo es una dirección física que siempre ha de ser divisible entre 16. Si quiere direccionar la memoria, el ordenador siempre ha de saber a qué segmento ha de acceder. El direccionamiento dentro de un segmento se realiza mediante el offset. El offset siempre tiene un tamaño de una palabra (Word, 2 bytes), ya que con una palabra se pueden direccionar exactamente 64 KBytes. Así que el acceso a la memoria siempre se realiza de la siguiente forma:

```
Destino := SEGMENTO:OFFSET.
```

Si programa en un lenguaje de alto nivel como por ejemplo Pascal, y se conforma con los tipos de datos predeterminados, no necesitará preocuparse más del direccionamiento de la memoria. Pascal se encargará automáticamente de ello. Pero si quiere entrar en profundidad en la programación del sistema, no podrá evitar el necesitar mayores conocimientos acerca de los diferentes modos de acceso a la memoria. El PC conoce, hasta el 286, cuatro registros de segmento. Por una parte está el segmento de código, que indexa el segmento en el que se encuentra el código que se va a ejecutar actualmente. El siguiente es el segmento de datos. Apunta a la zona de memoria en la que se guardan los datos del programa. El segmento extra le sirve al programador para operaciones de memoria o copia. El segmento de stack finalmente designa al stack (en castellano pila). A partir del 386 también se dispone de los registros *FS* y *GS*.

Hay distintas posibilidades de acceder a la memoria. Si programa en Pascal, seguramente conocerá el comando *MEM*. Mediante el comando

```
Resultado := MEM[Segmento:Offset];
```

puede copiar un byte de la memoria en una variable. Si por ejemplo quiere leer el píxel número 50 de la pantalla en modo MCGA, escriba:

```
Byte_50 := MEM[$A000:0050];
```

Si quiere realizar la misma acción en ensamblador, tenemos diferentes posibilidades para ello. El método más común, sobre todo para leer bloques de datos grandes, es el empleo de las instrucciones *lodsb/stosb/movsb* y sus equivalencias *WORD* y *DWORD*. Es algo más largo, ya que se ha de guardar el segmento de datos para poder acceder después a los datos del programa.

```
push ds
mov  si, 0a0000h
mov  ds, si
mov  si, 50
lodsb
mov  byte ptr byte_50, al
pop  ds
```

En la mayoría de los casos no querrá leer datos de un dispositivo de salida como la VGA, sino los que se encuentran en la memoria principal. No puede emplear aleatoriamente cualquier zona de datos, sino que ha de dejar que se le asigne la memoria necesitada. Esta tarea se la puede dejar a Pascal o al DOS. Este último método siempre es recomendable cuando se han de ejecutar otros programas en el ordenador, de forma paralela al suyo.

Los accesos de memoria a una de estas zonas reservadas se puede realizar mediante un puntero. Un puntero es un tipo de datos, que contiene el segmento y el offset de una zona de memoria. Normalmente se inicializan los punteros con el procedimiento *Getmem*, pero también se pueden construir. Mediante *Freemem* se puede volver a liberar la zona de memoria ocupada. La memoria ocupada por Pascal, sin embargo, sólo es accesible al programa de Pascal. Este lenguaje reserva al principio de la ejecución del programa la memoria mínima ajustada, y sólo la libera una vez que se haya finalizado el mismo.

Si no quiere emplear la memoria que Pascal pone a su disposición, sino que prefiere la memoria DOS convencional, debe emplear las funciones *48h* y *49h* de la inte-

rrupción DOS *21h*. Mediante la función *48h* puede reservar un bloque de memoria y mediante *49h* lo puede volver a liberar. Es importante saber que los bloques del DOS sólo se pueden ocupar en tamaños de párrafo. Esto significa que todas las indicaciones de tamaño se han de redondear al siguiente límite de 16 bytes. Los procedimientos de ensamblador *dos_getmem* y *dos_freemem* le muestran cómo se pueden realizar rutinas para el acceso a la memoria convencional.

```
dos_getmem proc pascal puntero:dword, cantidad:word
; ***************************************************************
; *** Aloja zona de memoria (máx. 64 KB) en la RAM del DOS    ***
; ***************************************************************
  push ds
  mov bx, cantidad
  shr bx, 4
  inc bx
  mov ah, 48h
  int 21h
  mov bx, w [puntero + 2]
  mov ds, bx
  mov bx, w [puntero]
  mov w [bx],0
  mov w [bx + 2],ax
  pop ds
  ret
dos_getmem endp

dos_freemem proc pascal puntero:dword
; ***************************************************************
; *** Vuelve a liberar zona memoria alojada con dos_getmem    ***
; ***************************************************************
  mov ax, word ptr [puntero + 2]
  mov es, ax
  mov ah, 49h
  int 21h
  ret
dos_freemem endp
```

11.2 EMS - un paso adelante para más memoria

Como la zona normal de memoria del DOS está limitada 640 KBytes, pero algunas aplicaciones cada vez necesitaban más memoria, los entonces líderes del mercado, Microsoft, Lotus e Intel se juntaron, para crear un nuevo estándar de memoria. El resultado fue el "Expanded Memory System", abreviado EMS. Originalmente se trataba de una tarjeta de expansión, que se podía conectar en los PC/XT, y que le "regalaban" al ordenador hasta 8 MBytes adicionales de memoria. Esta memoria se dividía en bloques de 64 KBytes, que se "visualizaban" en la zona normal de

direcciones. La zona "visualizada" se llama, en correspondencia a su función, como marco o ventana EMS o EMS-Frame. Los bloques "visualizados" se dividen en cuatro páginas (Pages) de 16 KBytes cada una y se pueden direccionar como memoria convencional.

El acceso a las páginas es regulado por el Expanded Memory Manager (abreviado: EMM). Así que no se ha de preocupar de los detalles técnicos. Con la Unit que presentamos en este libro obtiene una herramienta con la que podrá acceder a la EMS sin problemas, y sin tener que ocuparse más de los detalles específicos.

Las funciones del EMS

Las funciones del EMM se pueden acceder mediante la interrupción *67h*. Si aparece un error en una de sus funciones, el EMM devuelve un código de error, que se puede averiguar según la siguiente tabla:

Código	Significado
80h	Interfaz de software erróneo
81h	Hardware EMS erróneo
82h	EMM está ocupado
83h	Handle no válido
84h	Número de función no válido
85h	No quedan Handles disponibles
86h	Error al guardar o reponer la proyección entre páginas Físicas y lógicas
87h	No hay páginas libres
88h	Número de página no válido
89h	Se intentó ocupar 0 páginas
8Ah	Número de página no válido
8Bh	Número de página física incorrecto
8Ch	No se puede guardar el mapping
8Dh	Mapping ya está guardado
8Eh	Mapping no estaba guardado
8Fh	Número de subfunción incorrecto

Pero veamos ahora las funciones el EMM. Sólo hablaremos de las funciones a partir de la versión 3.0, ya que versiones más antiguas apenas están presentes por que su difusión fue muy poca.

Función 40h - Leer estado	
Entrada	AH = 40h
Salida	AH = Estado del EMM

Si la función devuelve el valor *0*, el EMM trabaja correctamente. De lo contrario el error se puede determinar según la tabla de errores.

Función 41h - Obtener dirección de segmento del Frame	
Entrada	AH = 41h
Salida	AH = Estado del EMM BX = Dirección de segmento de la ventana

La dirección de segmento sólo es válida si el código de estado contiene el valor *0*. De lo contrario hay un error según la tabla de errores. La dirección de segmento es la dirección en la que se puede encontrar el Frame en la memoria principal.

Función 42h - Obtener número de páginas EMS	
Entrada	AH = 42h
Salida	AH = Estado del EMM BX = Número de páginas EMS libres DX = Número de páginas EMS en total

Una página EMS mide 16 KBytes. La cantidad de memoria EMS libre resulta de una multiplicación de *16 * 1024*. Los valores sólo son válidos cuando el código de estado tiene el valor *0*.

Función 43h - Ocupar memoria EMS	
Entrada	AH = 43h BX = Número de páginas de 16 KB a ocupar
Salida	AH = Estado del EMM DX = Número de Handle

Las páginas alojadas se direccionan mediante un Handle, que sólo es válido si el código de estado contiene el valor *0*. Es importante que no pierda el Handle y que vuelva a liberar las páginas al finalizar el programa. De lo contrario se quedan bloqueadas para los siguientes programas. Se pueden ocupar un máximo de 512 páginas.

Función 44h - Fijar Mapping	
Entrada	AH = 44h AL = Número de la página en el Frame EMS BX = Número de la página en el bloque reservado DX = Handle del bloque reservado
Salida	AH = Estado del EMM

El número de la página en el EMS-Frame puede adoptar un valor entre 0 y 3. El número de la página en el bloque reservado se ha de entender relativa al inicio del bloque reservado (por ejemplo la sexta página del Handle 1Ch). La zona indicada se "visualiza" inmediatamente en el marco EMS del DOS. Si el código de estado contiene el valor 0, la función trabaja sin problemas.

Función 45h - Liberar páginas	
Entrada	AH = 45h DX = Handle
Salida	AH = Estado del EMM

Esta función libera de nuevo las páginas EMS reservadas. Se liberan todas las páginas del Handle. Llame siempre esta función al final del programa para todos los Handles ocupados por usted, ya que de lo contrario la memoria EMS ocupada se perdería para otros programas. El código de estado de 0 indica que la ejecución no ha presentado problemas.

Función 46h - Obtener versión del EMS	
Entrada	AH = 46h
Salida	AH = Estado del EMM AL = Número de versión

El número de versión está guardado en formato BCD. El número de versión principal se encuentra en los 4 bits superiores y el número de versión secundario en los 4 inferiores. Si el código de estado es diferentes de 0, ha aparecido un error.

Función 47h - Guardar Mapping	
Entrada	AH = 47h DX = Handle
Salida	AH = Estado del EMM

Mediante esta función se guarda la asociación del Mapping en el EMM como configuración. La función trabaja sin problemas, si el código de estado es igual a 0.

Función 48h - Liberar Mapping	
Entrada	AH = 48h DX = Número de Handle
Salida	AH = Estado del EMM

Con esta función se puede volver a liberar la asociación realizada mediante la función 47 Handle. Si el código de estado es distinto de cero, ha aparecido un error.

Función 4Bh - Obtener número de Handles	
Entrada	AH = 4Bh
Salida	AH = Estado del EMM BX = Número de los Handles ocupados actualmente

Esta función obtiene los Handles ocupados actualmente por el EMM. Un código de estado distinto de 0 indica un error.

Función 4Ch - Averiguar páginas ocupadas	
Entrada	AH = 4C DX = Número de Handle
Salida	AH = Estado del EMM BX = Número de páginas ocupadas

Con esta función puede averiguar cuántas páginas de 16 KBytes de memoria EMS ocupa el Handle pasado.

Función 4Dh - Averiguar Handles repartidos	
Entrada:	AH = 4Dh ES = Segmento de la dirección de la estructura de datos DI = Offset de la dirección de la estructura de datos
Salida:	AH = Estado del EMM BX = Número de páginas ocupadas en total

La función carga los números de todos los Handles activos con la cantidad de páginas ocupadas, en una estructura de datos. La estructura tiene la longitud número Handles * 4. Las entradas de la tabla se componen de dos WORD. En la primera palabra se guarda el número del Handle y en la segunda la cantidad de páginas.

El empleo del EMS

Vayamos ahora a por el uso de la EMS. Para comenzar se debería comprobar si realmente hay instalada EMS. Esto se puede averiguar, comprobando la identificación de la interrupción 67 Handle. Si allí se encuentra la cadena 'EMMXXX', hay un controlador EMS instalado y se puede continuar con la determinación de la versión. El saber la versión es sobre todo importante si se quieren emplear determinadas funciones, que aún no están implementadas en la versión 3.0. Aquí tiene un ejemplo de procedimiento de inicialización:

```
procedure Check_for_EMS;
var emsseg : word;
    emsptr : pointer;
    emshead : EMS_Header;
begin;
  asm
    mov ax, 3567h
    int 21h
    mov emsseg, es
  end;
  move(ptr(emsseg, 0)^,emshead, 17);
  if emshead.identificacion = 'EMMXXXX' then begin;
    EMS_Presente := true;
    asm
      mov ah, 40h        {Obtener estado del controlador EMS}
      int 67h
      cmp ah, 0
      jne @EMS_Vers_Error
      mov ah, 46h        { Obtener versión EMS }
      int 67h
      cmp ah, 0
      jne @EMS_Vers_Error
      mov bl, al
      shr al, 4
      mov bh, al         { bh = Vers.maj }
      or  bl, 0Fh        { bl = Vers.min }
      mov EMS_Version, bx
      jmp @EMS_Vers_Fin
@EMS_Vers_Error:
      mov EMS_Presente, 1
@EMS_Vers_Fin:
    end;
```

```
  end else begin;
    EMS_Presente := false;
  end;
end;
```

A continuación debería obtener la dirección en la que se ha de "visualizar" la ventana EMS en la memoria principal. Esto es importante ya que todos los accesos de memoria se refieren a esta dirección. Emplee para ello la función 41Handle del EMM:

```
Function Obtener_Segmento_EMS (VAR Segmento : word) : byte;
VAR hseg : word;
    fresultado : byte;
begin;
  asm
    mov ah, 41h
    int 67h
    cmp ah, 0
    jne @EMS_Segerm_Error
    mov hseg, bx
    mov fresultado, 0
    jmp @EMS_Segerm_Fin
@EMS_Segerm_Error:
    mov fresultado, ah
@EMS_Segerm_Fin:
  end;
  Segmento := hseg;
  EMS_Segmento_obtener := fresultado;
end;
```

Para determinar de cuánta memoria dispone, debería emplear la función para averiguar las páginas EMS. La función 42Handle devuelve tanto el número total de páginas disponibles, así como las páginas libres. A nosotros nos interesa sobre todo este último valor, que se guarda en la siguiente función que presentamos en la variable global Paginas_EMS_libres. Esta función sirve como función auxiliar para EMS_free.

```
Function Obtener_paginas_EMS : byte;
var fresultado : byte;
begin;
  asm
    mov ah, 42h
    int 67h
    cmp ah, 0
    jne @EMS_ObtPaginas_Error
    mov EMS_Paginas_Libres, bx
    mov EMS_Paginas_Insg, dx
    mov fresultado, 0
```

```
    jmp @EMS_ObtPaginas_Fin
@EMS_ObtPaginas_Error:
    mov fresultado, ah
@EMS_ObtPaginas_Fin:
  end;
  EMS_Obtener_paginas := fresultado;
end;
```

Si en su programa necesita el valor de la memoria EMS libre, debería emplear la función EMS_free. Devuelve el número de bytes libres que quedan en la EMS y emplea la función auxiliar Obtener_Paginas_EMS.

```
function EMS_free  : longint;
var    ayuda : longint;
begin;
  Obtener_Paginas_EMS;
  ayuda := Paginas_EMS_libres;
  EMS_free := ayuda SHL 14;
end;
```

Para poder alojar memoria EMS desde su programa, puede programar el EMM directamente. Es mucho más sencillo emplear la función *Getmem_EMS*. A ella le ha de pasar como primer parámetro, el Handle de destino bajo el cual el bloque ha de ser direccionable y como segundo parámetro el tamaño en bytes del bloque a ocupar. La función adapta su petición al número de páginas disponible y ocupa la memoria mediante la función *43h*.

```
Function Getmem_EMS(VAR H : EMSHandle; Size : longint) : byte;
var Fresultado : byte;
    EPaginas : word;
    Hhandle : word;
begin;
  EPaginas := (Size DIV 16384) + 1;
  asm
    mov ah, 43h
    mov bx, EPaginas
    int 67h
    cmp ah, 0
    jne @Getmem_Ems_Error
    mov Hhandle, dx
    mov fresultado, 0
    jmp @Getmem_Ems_Fin
@Getmem_Ems_Error:
    mov Fresultado, ah
@Getmem_Ems_Fin:
  end;
  H := Hhandle;
  Getmem_EMS := Fresultado;
end;
```

Si quiere volver a liberar la memoria EMS, y eso lo debería hacer siempre al final de su programa, simplemente emplee la función *Freemem_EMS*. Mediante el empleo de la función EMM *45Handle*, esta vuelve a liberar la zona de memoria que se gestionaba mediante el Handle pasado.

```
Function Freemem_EMS(H : EMSHandle) : byte;
var Fresultado : byte;
begin;
  asm
    mov ah, 45h
    mov dx, H
    int 67h
    mov Fresultado, ah
  end;
  Freemem_EMS := Fresultado;
end;
```

La función *Asignación_EMS* es muy importante para usted. Con ella puede determinar qué páginas se han de proyectar del EMS en la ventana EMS. Como primer parámetro le ha de pasar a la función el Handle bajo el que se direcciona el bloque de páginas. Como segundo parámetro se le ha de comunicar a la función qué páginas se han de ocupar en el ventana EMS (*0* hasta *3*). El número de la página en el EMS se ha de pasar como último parámetro.

```
Function Asignación_EMS(H : EMSHandle; PaginaPage, PaginaEMS : word)
: byte;
VAR Fresultado : byte;
begin;
  asm
    mov ah, 44h
    mov al, byte ptr PaginaPage
    mov bx, PaginaEMS
    mov dx, H
    int 67h
    mov Fresultado, ah
  end;
  EMS_Asignación := Fresultado;
end;
```

Cuando quiere guardar la asignación de las páginas EMS para un Handle determinado, emplee la función *EMS_guardar_asignación*. Tenga en cuenta que la asignación guardada no se puede modificar, hasta que no haya sido desbloqueada.

```
Function EMS_guardar_asignación(H : EMSHandle) : byte;
VAR Fresultado : byte;
begin;
  asm
```

```
      mov ah, 47h
      mov dx, H
      int 67h
      mov Fresultado, ah
    end;
    EMS_Guarda Asignación := Fresultado;
  end;
```

Mediante *EMS_desbloquear_asignacion* puede desbloquear de nuevo la asignación guardada de páginas EMS. La rutina emplea para ello la función EMM *48Handle*. Las asignaciones guardadas se han de desbloquear antes de poder ser modificadas.

```
Function EMS_desbloquear asignación(H : EMSHandle) : byte;
VAR Fresultado : byte;
begin;
  asm
    mov ah, 48h
    mov dx, H
    int 67h
    mov Fresultado, ah
  end;
  EMS_desbloquear_asignacion:= Fresultado;
end;
```

La función *RAM_2_EMS* ofrece un ejemplo de cómo puede descargar datos en la EMS. Con ella puede copiar un bloque de la RAM del DOS a la EMS. Tome esta función como ejemplo para sus propias rutinas, o empléelo directamente en la práctica. Si el tiempo le es muy crítico, debería formular su rutina especialmente para cada aplicación ya que el método aquí presentado sólo sirve para la descarga, pero no para el acceso directo a las páginas.

```
Function RAM_2_EMS(q : pointer; H : EMSHandle; Size : longint) :
byte;
VAR fresultado : byte;
    EMSseg     : word;
    hp         : ^byte;
    li         : word;
begin;
  EMS_Obtener segmento(EMSseg);
  hp := q;
  if Size > 16384 then begin;
    { Necesarias mas de una página }
    for li := 0 to (Size SHR 14)-1 do begin;
      EMS_Asignacion(H, 0, li);
      move(hp^, ptr(EMSseg, 0)^,16384);
      dec(Size, 16384);
```

```
      inc(hp, 16384);
    end;
    EMS_Asignacion(H, 0, li+1);
    move(hp^, ptr(EMSseg, 0)^,16384);
    dec(Size, 16384);
    inc(hp, 16384);
  end else begin;
    EMS_Asignacion(H,0,0);
    move(hp^, ptr(EMSseg, 0)^,16384);
    dec(Size, 16384);
    inc(hp, 16384);
  end;
end;
```

Con ayuda de la función *EMS_2_RAM*, puede volver a copiar de vuelta un bloque de la memoria EMS a la RAM del DOS.

```
Function EMS_2_RAM(q : pointer; H : EMSHandle; Size : longint) :
byte;
VAR fresultado      : byte;
    EMSseg          : word;
    hp              : ^byte;
    li              : word;
begin;
  EMS_Obtener segmento(EMSseg);
  hp := q;
  if Size > 16384 then begin;
    { Más de una página necesaria }
    for li := 0 to (Size SHR 14)-1 do begin;
      EMS_Asignación(H, 0, li);
      move(ptr(EMSseg, 0)^, hp^, 16384);
      dec(Size, 16384);
      inc(hp, 16384);
    end;
    EMS_Asignación(H, 0, li+1);
    move(ptr(EMSseg, 0)^, hp^, 16384);
    dec(Size, 16384);
    inc(hp, 16384);
  end else begin;
    EMS_Asignación(H,0,0);
    move(ptr(EMSseg, 0)^, hp^, 16384);
    dec(Size, 16384);
    inc(hp, 16384);
  end;
end;
```

Si en su programa necesita la información de cuantas páginas EMS están ocupadas por un Handle en particular, puede emplear la función *4Ch* del EMM. La función

EMS_paginas_ocupadas necesita como primer parámetro el número del Handle a analizar, como segundo la variable en la que se ha de guardar el número de páginas.

```
Function EMS_Paginas ocupadas (H : EMSHandle; var Paginas : word) :
byte;
var fresultado : byte;
    Hs : word;
begin;
  asm
    mov ah, 4Ch
    mov dx, H
    int 67h
    mov HS, bx
    mov fresultado, ah
  end;
  Paginas := Hs;
  EMS_paginas ocupadas := Fresultado;
end;
```

Una última función útil es la que obtiene el número de Handles EMS ocupados. Por desgracia no se puede disponer de un número cualquiera de Handles, sino que habitualmente están limitados a 32. Esta es también la razón de que siempre ha de procurar bloques lo más grandes posible y direccionarlos mediante un Handle.

```
Function EMS_asignar_handles(Var Numero : word) : byte;
Var Fresultado : byte;
    Han        : word;
begin;
  asm
    mov ah, 4Bh
    int 67h
    mov Han, bx
    mov Fresultado, ah
  end;
  Numero := Han;
  EMS_ asignar_handles := Fresultado;
end;
```

11.3 XMS - Y tuya sea la memoria...

El estándar XMS fue desarrollado por las compañías Microsoft, Lotus e Intel. En su creación también participó la empresa AST Research. A diferencia del estándar EMS, XMS no se basa en una tarjeta de ampliación de hardware. Las funciones del XMS se ponen a disposición por un controlador XMS, que se coloca en el interrupción *2Fh*. El controlador XMS más conocido es probablemente HIMEM.SYS que se incluye en el CONFIG.SYS y que se suministra junto con DOS o Windows.

Los códigos de error XMS

Las funciones del controlador no se direccionan mediante una interrupción, sino como Far Call. Para obtener la dirección del controlador ha de comprobar primero si hay instalado un controlador XMS. Esto lo puede hacer llamando la interrupción *2Fh* con el valor *4300h* en el registro *AX*. Si se devuelve el valor *80h* en el registro *AL*, el controlador está instalado. En ese caso la interrupción se ha de llamar de nuevo, con el valor *4310h* en el registro *AX*. Obtendrá la dirección del segmento del controlador en el registro *ES* y la dirección de offset del XMM en el registro *BX*.

Cuando haya obtenido la dirección del XMM (Extended Memory Manager), puede llamar sus funciones mediante un Far Call. Si la función se pudo procesar con éxito, habitualmente devuelve el código de estado *0001h* en *AX*. En ese caso el registro *BL* contendrá un código de error. Los códigos de error tienen el siguiente significado:

BL	Significado
80h	Función desconocida
81h	VDISK-Ramdisk encontrado
82h	Error en la línea de direcciones A20
8Eh	Error general de controlador
8Fh	Error irrecuperable
90h	HMA no presente
91h	HMA ya asignado
92h	Tamaño especificado en DX demasiado pequeño
93h	HMA no alojada
94h	Línea de direcciones A20 aún activa
A0h	No queda XMS disponible
A1h	Todos los Handles XMS ocupados
A2h	Handle no válido
A3h	Handle fuente no válido
A4h	Offset fuente no válido
A5h	Handle destino no válido
A6h	Offset destino no válido
A7h	Longitud incorrecta para función move
A8h	Solapado incorrecto en función move
A9h	Parity-Error
AAh	UMB no bloqueado
ABh	UMB aún bloqueado
ACh	Rebase del contador de bloqueo UMB
ADh	UMB no se puede bloquear
B0h	UMB disponible es más pequeño
B1h	No hay UMB disponible
B2h	Dirección de segmento UMB incorrecta

La programación XMS

El número de función de las funciones XMS se pasa siempre en el registro *AH*. Aquí les ofreceremos un pequeño resumen de las funciones básicas del XMM. Si las informaciones suministradas no le son suficientes, o desea una lista completa de todas las funciones XMS, le recomendamos literatura sobre el sistema, como por ejemplo PC Interno 2.0 de MARCOMBO, S.A.

Función 00h - Obtener número de versión	
Entrada	AH = 00h
Salida	AX = número de versión XMS BX = número de revisión interno DX = Estado de la HMA

Con esta función puede averiguar el número de versión del controlador. Si en el estado HMA se encuentra el valor 1, la HMA está disponible, de lo contrario no se permite el acceso a la misma.

Función 03h - Activar A20 globalmente	
Entrada	AH = 03h
Salida	AX = código de error

Si quiere emplear la HMA en el modo real de forma fiable, ha de llamar esta función antes o después de la petición de la HMA. Recuerde cerrar la línea de direcciones antes de finalizar su programa, para evitar rebases de segmento en los programas subsiguientes.

Función 04h - Cerrar A20 globalmente	
Entrada	AH = 04h
Salida	AX = código de error

Con esta función cierra de nuevo la línea de direcciones *A20*.

Función 05h - Activar A20 localmente	
Entrada	AH = 05h
Salida	AX = código de error

Mediante esta función se activa la línea de direcciones *A20*. Es decir, que sólo se activa si la función no había sido llamada con anterioridad. Esta comprobación se realiza mediante un contador de llamadas, que es incrementado por la función *05h* y decrementado por la función *06h*.

Función 06h - Desactivar A20 localmente	
Entrada	Ah = 06h
Salida	AX = Código de error

La opuesta a la función *05h* desactiva localmente la línea de direcciones *A20*.

Función 07h - Obtener estado de A20	
Entrada	AH = 07h
Salida	AX = estado de la línea de direcciones

Si *A20* está libre, obtiene el valor *0001h* en *AX*, de lo contrario será *0000h*.

Función 08h - Obtener XMS libre	
Entrada:	AH = 08h
Salida:	AX = Longitud del mayor bloque libre DX = tamaño total del XMS en KBytes

Con esta función puede averiguar la memoria XMS libre. Pero atención, los valores devueltos siempre miden 64 KBytes de más, ya que también se cuenta la HMA, que también mide 64 KBytes.

Función 09h - Alojar bloque en la memoria extendida	
Entrada	AH = 09h DX = Tamaño del bloque en KBytes
Salida	AX = Código de error DX = Handle para el acceso al bloque

Esta función reserva un bloque de memoria extendida (EMB) en XMS. El alojamiento sólo funciona si aún queda libre un bloque del tamaño suficiente en la XMS. El bloque ocupado se puede direccionar a través del Handle devuelto.

Función 0Ah - Liberar bloque de memoria extendida	
Entrada	AH = 0Ah DX = Handle
Salida	AX = Código de error

Mediante esta función vuelve a liberar un EMB alojado. Después de la llamada de la función, el Handle queda como no válido, y los datos están permanentemente borrados.

Función 0Bh - Copiar memoria	
Entrada	AH = 0Bh DS = Segmento de la estructura de copia SI = Offset de la estructura de copia
Salida	AX = Código de error

Con esta función puede copiar memoria a/de la XMS. Ya que los registros no bastarían como para guardar todos los datos, la función emplea una estructura de copia. Tiene la siguiente forma:

Offset	Función	Tipo de datos
00h	Longitud del bloque a copiar	1 Dword
04h	Handle del bloque fuente	1 Word
06h	Offset en el bloque fuente,	a partir de aquí se copia 1 Dword
0Ah	Handle del bloque destino	1 Word
0Ch	Offset en el bloque destino,	a partir de aquí se copia 1 Dword

Cuando se presenta el solapado de las zonas, la zona fuente ha de estar antes de la zona destino, sino no se garantiza un funcionamiento perfecto. Para permitir una copia rápida, debería comenzar sus zonas de memoria en una dirección divisible entre cuatro.

Función 0Ch - Bloquear EMB contra desplazamiento	
Entrada	AH = 0Ch DX = Handle
Salida	AX = Código de error DX:BX = La dirección lineal de 32 bits del EMB

Para que no se creen agujeros en la XMS, el XMM, si es necesario, desplaza algunos bloques de memoria. Para asegurar un bloque contra el desplazamiento, lo que puede ser interesante si se quiere acceder directamente a la memoria, se emplea esta función. Como resultado obtiene en *DX:BX* la dirección lineal bajo la cual puede direccionar la memoria.

Función 0Dh - Desbloquear EMS asegurado	
Entrada	AH = 0Dh DX = Handle
Salida	AC = Código de error

Mediante esta función puede volver a liberar un bloque asegurado mediante la función *0Ch*.

Función 0Eh - Obtener informaciones sobre el EMB	
Entrada	AH = 0Eh DX = Handle
Salida	AX = Código de error DX = longitud del EMB

Mediante estas funciones se pueden obtener tanto informaciones sobre el bloque de memoria, como determinar el número de Handles XMS libres. Si la función ha trabajado sin errores, en *AX* se puede encontrar el valor *0001h*. En ese caso se encuentra el *BH* el contador de bloqueo del bloque *EM*. El contador de bloqueo se incrementa después de cada llamada con la función *0Ch* y se decrementa de nuevo con la función *0Dh*. Si el valor de BH es mayor que *0*, ha de llamar a la función *0Dh BH* veces, para liberar al bloque. En *BL* se encuentra la cantidad de Handles XMS libres. Tenga en cuenta que es mejor ocupar bloques de memoria lo más grande posibles, ya que el número de Handles XMS está limitado. Si en AX se puede encontrar el valor *0000h*, existe un error. La causa del error se puede determinar con ayuda de la tabla de errores. En caso de una ejecución con éxito se encuentra en *DX* la longitud del EMB en KBytes.

Función 0Fh - Modificar tamaño EMB	
Entrada	AH = 0Fh BX = nuevo tamaño del EMB DX = Handle
Salida	AX = Código de error

Para que no se creen agujeros en la XMS, el XMM, si es necesario, desplaza algunos bloques de memoria. Para asegurar un bloque contra el desplazamiento, lo que puede ser interesante si se quiere acceder directamente a la memoria, se emplea esta función. Como resultado obtiene en *DX:BX* la dirección lineal bajo la cual puede direccionar la memoria.

Función 0Dh - Desbloquear EMS asegurado	
Entrada	AH = 0Dh DX = Handle
Salida	AC = Código de error

Mediante esta función puede volver a liberar un bloque asegurado mediante la función *0Ch*.

Función 0Eh - Obtener informaciones sobre el EMB	
Entrada	AH = 0Eh DX = Handle
Salida	AX = Código de error DX = longitud del EMB

Mediante estas funciones se pueden obtener tanto informaciones sobre el bloque de memoria, como determinar el número de Handles XMS libres. Si la función ha trabajado sin errores, en *AX* se puede encontrar el valor *0001h*. En ese caso se encuentra el *BH* el contador de bloqueo del bloque *EM*. El contador de bloqueo se incrementa después de cada llamada con la función *0Ch* y se decrementa de nuevo con la función *0Dh*. Si el valor de BH es mayor que *0*, ha de llamar a la función *0Dh BH* veces, para liberar al bloque. En *BL* se encuentra la cantidad de Handles XMS libres. Tenga en cuenta que es mejor ocupar bloques de memoria lo más grande posibles, ya que el número de Handles XMS está limitado. Si en AX se puede encontrar el valor *0000h*, existe un error. La causa del error se puede determinar con ayuda de la tabla de errores. En caso de una ejecución con éxito se encuentra en *DX* la longitud del EMB en KBytes.

Función 0Fh - Modificar tamaño EMB	
Entrada	AH = 0Fh BX = nuevo tamaño del EMB DX = Handle
Salida	AX = Código de error

```
    xms_long := xms_in_kb;
    XMS_free := xms_long * 1024;
  end;
```

Ahora que ya sabe de cuánta memoria dispone, la puede ocupar. Emplee para ello la función *Getmem_XMS*. Se orienta en la función de Pascal *Getmem*. Como primer parámetro ha de pasarle la variable en la que se debe guardar al Handle para el acceso a la memoria alojada. El segundo parámetro designa el tamaño del bloque a reservar en bytes.

```
Function Getmem_XMS(VAR H : XMSHandle; Size : longint) : byte;
var bsize : word;
    Fresult : byte;
    xmsh : word;
begin;
  bsize := (size DIV 1024) + 1;
  asm
    mov ax, 0900h                    { 9 = Alojar zona de memoria }
    mov dx, bsize
    call dword ptr [XMST]
    cmp ax, 1
    jne @Error_GetmemXms
    mov xmsh, dx
    mov Fresult, 0
    jmp @Fin_GetmemXms
@Fehler_GetmemXMS:
    mov Fresult, bl
@Ende_GetmemXms:
  end;
  h := xmsh;
  Getmem_Xms := Fresult;
end;
```

Al final del programa debería liberar toda la memoria XMS ocupada. Esto se puede hacer con la función *Freemem_XMS*. Como parámetros, la función necesita simplemente el número del Handle del EMB.

```
Function Freemem_XMS(H : XMSHandle) : byte;
var fresult : byte;
begin;
  asm                                { A = Liberar zona de memoria }
    mov ax, 0a00h
    call dword ptr [XMST]
    cmp ax, 1
    jne @Error_FreememXms
    mov Fresult, 0
```

```
    jmp @Fin_FreememXms
@Fehler_FreememXms:
    mov Fresult, bl
@Ende_FreememXms:
  end;
end;
```

Toda la memoria alojada no nos sirve de nada, sino podemos copiar datos a ella o de ella. Por ello ahora les presentamos las funciones *Ram_2_XMS* y *XMS_2_Ram*. Con la primera puede copiar datos de la memoria principal a la XMS, la segunda función hace el camino inverso. tenga en cuenta que primero ha de alojar memoria XMS mediante la función *Getmem_XMS*, antes de poder copiar datos a ella.

```
Function RAM_2_XMS(q : pointer; h : XMSHandle; Size : Word) : byte;
VAR fresult : byte;
begin;
  XC.Size     := Size;
  XC.Q_Handle := 0;                { 0 = RAM }
  XC.Q_Offset := q;
  XC.Z_Handle := h;
  XC.Z_Offset := nil;
  asm
    mov si,offset XC
    mov ax,0B00h
    call dword ptr [XMST]
    cmp ax,1
    jne @Error_RAM2XMS
    mov fresult,0
    jmp @Fin_RAM2XMS
@Error_RAM2XMS:
    mov fresult,bl
@Fin_RAM2XMS:
  end;
end;

Function XMS_2_Ram(d : pointer; h : XMSHandle; Size : Word) : byte;
VAR fresult : byte;
begin;
  XC.Size     := Size;
  XC.Q_Handle := h;
  XC.Q_Offset := nil;
  XC.Z_Handle := 0;                { 0 = RAM }
  XC.Z_Offset := d;
  asm
    mov si,offset XC
    mov ax,0B00h
    call dword ptr [XMST]
```

```
    cmp ax,1
    jne @Error_XMS2RAM
    mov fresult,0
    jmp @Fin_XMS2Ram
@Error_XMS2Ram:
    mov fresult,bl
@Fin_XMS2Ram:
  end;
end;
```

Como última función, *XMS_2_XMS* le ofrece la posibilidad de copiar datos dentro de la misma XMS. Esto puede ser interesante si quiere cargar por ejemplo una imagen del disco, y después duplicarla. Esta función necesita como primer parámetro el Handle del bloque fuente, como segundo parámetro el Handle del bloque destino. El último parámetro indica el tamaño en bytes.

```
Function XMS_2_XMS(h1, h2 : XMSHandle; Size : Word) : byte;
VAR fresult : byte;
begin;
  XC.Size      := Size;      { tamaño del bloque en bytes }
  XC.Q_Handle  := h1;        { handle fuente }
  XC.Q_Offset  := nil;       { Offset fuente, 0 = inicio del bloque}
  XC.Z_Handle  := h2;        { handle destino }
  XC.Z_Offset  := nil;       { offset destino }
  asm
    mov si, offset XC
    mov ax, 0B00h
    call dword ptr [XMST]
    cmp ax, 1
    jne @Error_RAM2XMS
    mov fresult, 0
    jmp @Fin_Ram2XMS
@Fehler_Ram2XMS:
    mov fresult, bl
@Ende_Ram2XMS:
  end;
end;
```

11.4 El modelo de memoria Flat - la solución de sus problemas de memoria

El empleo de XMS será una solución aceptable para el problema de memoria en muchos casos, pero presenta un pequeño inconveniente: no se puede acceder directamente a la XMS, sino que siempre se ha de copiar a/de la XMS en forma de bloques. Esto no sólo es molesto, sino que sobre todo consume tiempo.

Así que si ha de acceder directamente a la memoria, o si necesita velocidad, ha de emplear el modelo Flat. El modelo Flat le ofrece la posibilidad de acceder linealmente a toda la memoria del PC, y es soportado por el nuevo Borland Pascal 8.0. Como usuario de una versión más pequeña, tendrá que ayudarse a sí mismo.

La idea del modelo Flat se basa en el siguiente truco: El procesador (al menos ha de ser un 386) se pasa al protected Mode. Allí se adaptan los límites de segmento para los registro *FS* y *GS* a 4 GBytes. Ahora se vuelve a pasar al procesador al modo real sin reiniciarlo. Con ello disponemos de 4 GBytes teóricos. Esto debería bastar para comenzar.

El trasfondo técnico del modelo Flat

¿Cómo funciona todo ello a nivel técnico? Primero nos habremos de dedicar un poco a la gestión de memoria en el Protected Mode. En el Protected Mode no se direcciona la memoria directamente, sino a través de los llamados *selectores*. Los selectores (*to select*, inglés, seleccionar) son puntero a un *Descriptor* (*to describe*, inglés, describir). En estos descriptores se guardan todas las informaciones acerca del segmento correspondiente: su tamaño, su dirección física y sus derechos de acceso. De forma estándar, el tamaño de un segmento se determina en *0FFFFh* al encender el ordenador. Estos son los mágicos 65.536 bytes a los que puede acceder normalmente bajo DOS. En el Protected Mode no está sometido a este límite. Puede direccionar un máximo de 4 GBytes de memoria. Sólo ha de adaptar adecuadamente el descriptor para el segmento correspondiente. Esta adaptación se regula mediante la *Global Descriptor Table* (tabla global de descriptores), abreviada GDT. En ella se encuentran las informaciones sobre los segmentos.

Si quiere conmutar al modelo Flat, primero ha de construir una de estas GDT. La GDT en principio se ha de entender como un array de descriptores. Un descriptor tiene la siguiente estructura:

Número bits	Significado
16 Bits	Longitud del segmento
24 Bits	Dirección física
8 Bits	Derechos de acceso
16 Bits	Bits adicionales para longitud de segmento

El GDT primero ha de contener un segmento nulo para nuestros propósitos (todos los valores = *0*). A continuación sigue el segmento modificado para el acceso hasta 4 GBytes. Los 16 bits bajos se ponen a *0FFFFh*, los 16 bits altos de la longitud de segmento contienen el valor *0FFCFh*. La dirección física la ponemos a *0*. Con

ello el segmento comienza al principio de la memoria. Los derechos de acceso se distribuyen mediante un valor de *92h* con la más alta prioridad para un segmento de datos.

Después de habernos construido un GDT de esta forma, lo hemos de cargar mediante la instrucción *lgdt*. Después de haber enmascarado todas la interrupciones mediante *cli*, para que nadie se nos meta en medio, se puede conmutar al Protected Mode. Allí ha de borrar inmediatamente la Execution Pipe del procesador mediante un salto. Esto es importante, ya que sino la máquina se quedará colgada, porque aún tiene instrucciones en la Prefetch-Queue. Adapte ahora los segmentos a 4 GBytes, equipando a los segmentos con el selector correspondiente. Después de haber realizado esto, puede volver a conmutar al modo real, sin reiniciar el procesador. También aquí se ha de realizar un salto inmediatamente después de la conmutación. No olvide para finalizar, el volver a activar las interrupciones.

Así se programa el modelo Flat

Pero basta de teoría, aquí tiene un procedimiento de ensamblador con el que puede inicializar el modelo Flat. La GDT está declarada externamente.

```
;#################################################################
;###                                                           ###
;###          pasa al procesador al modelo flat                ###
;###                                                           ###
;#################################################################
Enable_4Giga proc pascal
    mov GDT_Off[0],16
    mov eax,seg GDT
    shl eax,4
    mov bx,offset GDT
    movzx ebx,bx
    add eax,ebx
    mov dword ptr GDT_Off[2],eax
    lgdt pword ptr GDT_Off          ; cargar GDT
    mov bx,08h                      ; bx apunta a la 1ª entrada del GDT
    push ds
    cli                             ; apagar interrupciones
    mov eax,cr0                     ; pasar al Protected Mode
    or eax,1
    mov cr0,eax
    jmp In_den_Protectedmode        ; borrar Executionpipe
    In_den_Protectedmode:
    mov gs,bx                       ; adaptar segmentos a 4 GB
```

```
        mov fs,bx
        mov es,bx
        mov ds,bx
        and al,0FEh                     ; volver al modo real, sin
        mov cr0,eax                     ; resetear el procesador
        jmp In_den_Realmode             ; borrar Executionpipe
        In_den_Realmode:
        sti                             ; activar interrupciones
        pop ds
        ret
    Enable_4Giga endp

    code ends

    END
```

A continuación necesitamos un procedimiento con el que podamos acceder a la memoria. Por desgracia la *RMEM* (Real Memory) no se puede direccionar mediante los comandos *lodsb/stosb/movsb*. Sino que la *RMEM* se ha de direccionar mediante su dirección física. Para ello se guarda la dirección deseada en una variable *longint*, mediante la que se puede acceder a la memoria. El acceso puede tener el siguiente aspecto:

```
mov ebx, Posición de variable
mov al, gs:[ebx]
```

Antes de acceder a la memoria, ha de asegurarse de que el 20º bit de direcciones está libre. Normalmente suele provocar que se conmute al inicio de la memoria. Si se accede a la memoria extendida con, por ejemplo, controladores *XMS*, el controlador se ocupa él mismo de ello. Pero hemos de liberar este bit explícitamente. Una programación directa está unida a un cierto esfuerzo, ya que se ha de hacer servir el controlador del teclado. Pero como de todas formas ya empleamos el controlador *HIMEM*, podemos emplear las funciones que este pone a nuestra disposición.

Un ejemplo, de cómo se puede realizar el acceso a la *RMEM* en la práctica, lo puede encontrar en el procedimiento *mem_write*. Con él puede copiar un bloque de la memoria principal a la *RMEM*. Como primer parámetro necesita el destino en la *RMEM*. Como segundo, ha de indicarle el offset del puntero a la zona fuente de la memoria principal, seguido del segmento del puntero. El último parámetro representa a la longitud del bloque a copiar.

```
;#################################################################
;###                                                           ###
;###      Copiar un bloque de la memoria principal al RMEM     ###
;###                                                           ###
;#################################################################

mem_Write proc pascal pFuente:dword, seg_ofs:word, seg_destino:word,
longitud:word
    call xms_Enable_A20
    mov  ax,seg_destino          ; Addy de mem. principal a ES:SI
    mov  es,ax
    mov  di,seg_ofs
    xor  ax,ax                   ; direcc. fuente RMEM a GS:EAX
    mov  gs,ax
    mov  eax,pFuente
    mov  cx,longitud
nloop:
    mov  bl,es:[di]              ; copiar bytes
    mov  byte ptr gs:[eax],bl
    inc  eax
    inc  di
    loop nloop
    ret
mem_Write endp
```

La rutina complementaria la representa el procedimiento *mem_read*. Con él puede copiar un bloque de la *RMEM* a la memoria principal. Como primer parámetro se le ha de pasar la posición fuente en la *RMEM*, como segundo el offset y a continuación el segmento del puntero a la zona de destino de la memoria principal. El último parámetro indica la longitud a copiar en bytes. Como zona de destino también se puede indicar la RAM de vídeo de la tarjeta VGA.

```
;#################################################################
;###                                                           ###
;###      copia un bloque del RMEM a la memoria principal      ###
;###                                                           ###
;#################################################################

mem_Lesen proc pascal pFuente:dword,seg_ofs : word,seg_destino :
word,longitud:word
    call xms_Enable_A20
```

```
    mov  ax,seg_destino          ; Addy de mem. principal a ES:SI
    mov  es,ax
    mov  di,seg_ofs
    xor  ax,ax                   ; dirección fuente RMEM a GS:EAX
    mov  gs,ax
    mov  eax,pFuente
    mov  cx,longitud
lloop:  mov  bl,byte ptr gs:[eax]  ; copiar byte

    mov  es:[di],bl
    inc  eax
    inc  di
    loop lloop
    ret
mem_Lesen endp
```

Es algo laborioso emplear cada vez el ensamblador Inline para cada llamada. Además sólo se trata de puros procedimientos de copia, sin comprobación de rangos. Por ello aquí presentamos dos pequeños procedimientos en Pascal, que hacen un juego de niños el acceso a la *RMEM*.

```
 ################################################################
 ###                                                          ###
 ###      Copia un bloque del RMEM a la memoria principal     ###
 ###                                                          ###
 ################################################################
}
begin
 if Fuente + longitud < Rmem_Max then begin
    Segm:=seg(Destino^);
    Offs:=ofs(Destino^);
    inc(Segm,Offs div 16);
    Offs:=Offs mod 16;
    inc(Fuente,MByte1);
    leer_mem(Fuente,Offs,Segm,longitud);
  end else begin;
    asm mov ax,0003; int 10h; end;
    writeln('Error reading back XMS Realmemory !');
    writeln('System halted');
    halt(0);
  end;
end;

procedure Rmem_write(Fuente:pointer;Destino:longint;longitud:word);
{
 ################################################################
 ###                                                          ###
 ###      Copia un bloque de la memoria principal al RMEM     ###
 ###                                                          ###
 ################################################################
```

```
}
begin
 if Destino+longitud < Rmem_Max then begin
   Segm := seg(Fuente^);
   Offs := ofs(Fuente^);
   inc(Segm,Offs div 16);
   Offs := Offs mod 16;
   inc(Destino,MByte1);
   mem_write(Destino, Offs,Segm,longitud);
  end else begin;
    asm mov ax,0003; int 10h; end;
    writeln('XMS allocation error ! Not enough memory ?');
    writeln('System halted');
    halt(0);
  end;
end;
```

Antes de emplear los procedimientos *RMEM*, se ha de asegurar en cualquier caso de que realmente exista la memoria correspondiente. Lo peor que le puede pasar a la Unit es que ya haya instalado un administrador de memoria como *EMM386*, *QEMM* o similares. La Unit no es compatible con ellos, ya que trabajan según un principio similar. Con la función *Multitasker_activo* puede comprobar con facilidad si hay activo uno de estos programas.

```
public Multitasker_activo

;*****************************************************************
;***                                                           ***
;***   Comprueba si hay presente un multitasker como EMM386    ***
;***                                                           ***
;*****************************************************************
Multitasker_activo proc near
   mov eax, cr0
   and ax, 1
   ret
Multitasker_activo endp
```

La comprobación de si hay suficiente memoria disponible, la puede realizar con el procedimiento *Memory_Checks*. Comprueba la memoria principal disponible y la memoria XMS. Para ello emplea el controlador *HIMEM*.

```
procedure memory_checks(minmain, minxms : word);
{
 *****************************************************************
 ***                                                           ***
 ***   Comprueba si hay suficiente memoria disponible          ***
 ***                                                           ***
 *****************************************************************
```

```
}
var xmsfree, mainfree : word;
begin;
  { Obtener memoria XMS libre }
  xmsfree := xms_free;
  { Obtener memoria principal }
  mainfree := memavail div 1024;
  { mensaje si no hay suficiente memoria libre }
  if (xmsfree < minxms) or (mainfree < minmain) then begin;
    asm mov ax, 0003; int 10h; end;
    writeln('Sorry, not enough memory available !');
    writeln('          You need    Available');
    writeln('XMS :      ',minxms :6,' KB          ',xmsfree:4,' KB');
    writeln('Main:      ',minmain:6,' KB          ',mainfree:4,' KB');
    halt(0);
  end;
end;
```

Para gestionar el *RMEM* se debería construir una Unit de gestión propia. La solución que aquí planteamos sólo tiene sentido si sabe con bastante exactitud qué hace con su memoria. Ese será el caso cuando programe una demo. Pero si pone a disposición de usuarios no demasiado expertos el manejo de la *RMEM*, debería pensar un poco en las rutinas *Freemem* y *Ordena*.

```
function Rgetmem(Var rpos : longint; rsize : longint) : boolean;
{
 ****************************************************************
 ***                                                          ***
 ***   Un procedimiento Getmem simplificado para la RMEM      ***
 ***                                                          ***
 ****************************************************************
}
begin;
  if Rmemposi + rsize > Rmem_max then begin;
    Rgetmem := false;
  end else begin;
    rpos := Rmemposi;
    inc(Rmemposi, rsize);
    Rgetmem := true;
  end;
end;
```

Finalmente queremos ofrecerle dos procedimientos con los que su sistema puede conmutar desde Pascal al modo *RMEM* y de vuelta. Sólo ha de pasarle la memoria *RMEM* deseada al procedimiento *enable_Realmem* en KBytes, el resto lo hace el procedimiento por sí mismo. Comprueba si hay instalado un controlador HIMEM,

si hay suficiente memoria y si hay activo un posible Multitasker. A continuación ocupa la memoria necesaria y conmuta al modo *RMEM*.

```
procedure enable_Realmem(Min : word);
{
 ################################################################
 ###                                                          ###
 ###                  Pasa al modo RMEM                       ###
 ###     Ha de haber «MIN» kbytes de memoria libre!           ###
 ###                                                          ###
 ################################################################
}
begin
 { ver si hay multitasker ... }
 if multitasker_activo then begin;
   asm mov ax,0003; int 10h; end;
   writeln('Processor already in V86 mode !');
   writeln('Please reboot without any EMS-drivers such as EMM386,
QEMM etc.');
   writeln('HIMEM.SYS is required ! ');
   halt(0);
 end;
 { controlador XMS instalado  }
 if not XMS_Presente then begin;
   asm mov ax,0003; int 10h; end;
   writeln('No XMS or Himem-driver available');
   writeln('Please reboot your System using HIMEM.SYS !!!');
   halt(0);
 end;
 { ocupar memoria necesitada }
 error := Getmem_XMS(My_XmsHandle,min*1024);
 if error <> 0 then begin;
   asm mov ax,0003; int 10h; end;
   writeln('Error during memory-allocation !');
   writeln('We need at least ',Min,' KB of free XMS Memory !!!');
   writeln('Please reboot your System using HIMEM.SYS');
   writeln;
   halt(0);
 end;
 { obtener posición física inicial }
 Rmemposi := XMS_lock(My_XmsHandle);
 if rmemposi < 1000000 then begin;
   asm mov ax,0003; int 10h; end;
   writeln('Error during memory-fixing !');
   writeln('We need at least ',Min,' KB of free XMS Memory !!!');
   writeln('Please reboot your System using HIMEM.SYS');
   writeln;
```

```
    halt(0);
  end;
  { Liberar }
  Enable_4Giga;
end;
```

Naturalmente debería "recoger" cuando termine. Es decir, liberar la memoria XMS empleada, ya que sino no estará disponible para los demás programas. Esto se puede realizar mediante el procedimiento *Exit_Rmem*. Este procedimiento *se ha* de llamar al final del programa, si ha empleado *RMEM*.

```
procedure Exit_Rmem;
{
 ******************************************************************
 ***                                                            ***
 ***   Exit-Procedure del RMEM, SE HA de llamar!                ***
 ***                                                            ***
 ******************************************************************
}
begin;
  { Liberar bloque }
  XMS_unlock(My_XmsHandle);
  { Liberar memoria }
  Freemem_XMS(My_XmsHandle);
end;
```

12. Sáquele partido a su ordenador - programación de los componentes auxiliares

En este capítulo podrá aprenderlo todo sobre los diferentes componentes auxiliares como el chip del temporizador, el controlador de interrupciones, el chip del reloj de tiempo real (RTC), así como el controlador DMA. Con todo ello estará en disposición de emplear a fondo el hardware de su ordenador.

12.1 El concepto de interrupciones

Una característica de los PC son las llamadas interrupciones. Las interrupciones son, como su nombre indica, interrupciones del sistema. Se disparan por las circunstancias más variadas. Hay interrupciones que son iniciadas por el sistema, el hardware instalado, otras por el software o el sistema operativo y finalmente aquellas que son generadas por hardware externo. Hemos de distinguir por las interrupciones enmascarables y no enmascarables, llamadas NMI. Más abajo puede encontrar más informaciones sobre ello. Este apartado se ocupará de los principios básicos de la programación de interrupciones y sus diferentes posibilidades de aplicación.

Cuando se llama una interrupción, se interrumpe el procesamiento del programa actual. Se ejecuta un trozo de código definido para la interrupción. Este termina con la instrucción de ensamblador *iret*, lo que representa a *Interrupt Return*. Durante la llamada de la interrupción se guardan en el stack (la pila) el registro de banderas, *CS* e *IP*. Así puede averiguar fácilmente en la interrupción, desde qué posición del programa principal fue disparada la interrupción. Si trabaja con Pascal, no se enterará de todas estas finezas. Pascal pone a su disposición algunos procedimientos, con los que puede manejar las interrupciones con sencillez.

En la memoria principal existe una tabla en la que se guardan las direcciones de los procedimientos que se llaman cuando se dispara la interrupción correspondiente. Estas direcciones no son nada más que punteros. Estos punteros ya los conocerá del manejo cotidiano de Pascal. Si quiere asignarle, por ejemplo, la dirección de su procedimiento a un puntero, lo puede realizar con la instrucción

```
My_Pointer := ADR(El_procedimiento);
```

o más breve

```
My_Pointer := @El_procedimiento;
```

Este simple manejo es soportado por Pascal durante la gestión de interrupciones. Para determinar la dirección de una interrupción determinada, por ejemplo la del temporizador, simplemente ha de introducir el comando

```
VAR  Pointer_al_Timer : pointer;
GetIntvec(8, Pointer_auf_Timer);
```

Para desviar una interrupción a su propia rutina, introduzca:

```
SetIntvec(8, @Mi_procedimiento_de_temporizador);
```

¡Ahora algo importante! Detrás de declaración de procedimientos de la rutina de interrupciones ha de escribir por fuerza la palabra clave *interrupt*. Esto es importante para que Pascal ejecute la instrucción *iret* al final de la rutina de interrupción y no la instrucción *ret* normal. La declaración podría tener el siguiente aspecto:

```
Procedure Mi_procedimiento_de_temporizador; Interrupt;
begin;
  writeln('¡Hola! Esta es una interrupción!');
  port[$20] := $20;
end;
```

Tenga en cuenta que la interrupción *8* es una interrupción de hardware. Al final de una de estas interrupciones ha de comunicarle al controlador de interrupciones que la interrupción se procesó hasta el final. Para ello envíe la instrucción *EOI* (End of Interrupt = 20h) al controlador que se encuentra en el port *20h*.

"¿Y todo esto para qué?" se preguntará ahora. Bien, podemos explicárselo con ayuda de la interrupción del temporizador. Quiere escribir un programa que cada medio segundo visualice el carácter ASCII #1 y cada medio segundo el carácter ASCII #2. Naturalmente puede comprobar en un bucle la hora del sistema y visualizar cada medio segundo un carácter. Pero esto impide el procesamiento de otras partes del programa, como por ejemplo la ordenación simultánea de un lista de 10.000 entradas.

La otra posibilidad es la de desviar la interrupción del temporizador, que se llama 18,2 veces por segundo. En esta rutina puede ir incrementando un contador, que controle cuántas veces se llamó la interrupción. Cada novena interrupción puede visualizar un carácter y reponer el contador. Ahora puede ordenar tranquilamente sus 10.000 entradas, visualizar el COMMAND.COM al revés o hacer sonar la escala tonal hacia arriba y abajo a la vez. Sus caracteres se visualizarán con la precisión de un reloj suizo. Sólo no ha de olvidar de reponer la interrupción a su rutina

original al final de su programa, rutina que obtuvo con *GetIntvec*. El intento de depurar paso a paso un programa controlado por interrupciones habitualmente termina de forma lamentable.

Ahora que ya sabe cómo puede programa interrupciones en Pascal, vamos a profundizar un poco en la materia. la pregunta es: ¿Qué hacen realmente los procedimientos *GetIntvec* y *SetIntVec*? Emplean las funciones *25h* y *35h* de la interrupción *21h* del DOS. Y ya hemos llegado a un nuevo fenómeno: una interrupción no sólo puede tener una rutina, sino también varias subfunciones. La interrupción *21h* es empleada por el DOS para todas las funciones importantes del sistema operativo. Para indicarle a la interrupción qué subfunción quiere ejecutar, no hay manera de evitar el ensamblador (Inline). Es necesario escribir el número de la función de interrupción deseada en el registro *AH*, y otros posibles parámetros en otros registros, lo que cambia de una subfunción a otra. En la función *25h* ha de inicializar los siguientes registros:

Registro	Contenido/Significado
AH	25h
AL	Número de la interrupción a fijar
DS	Dirección de su procedimiento de interrupción
DX	Dirección de offset de su procedimiento de interrupción

En la función *35h* no sólo ha de indicar valores, sino que también recibe valores de vuelta. Al fin y al cabo sirve para averiguar la dirección de la rutina de interrupción actual, y en algún lugar deberá averiguar el resultado de la consulta. Aquí tiene la ocupación de los registros:

Entrada	AH	35h
	AL	Número de la interrupción a averiguar
Salida	ES	Dirección de segmento de la rutina
	BX	Dirección de offset de la rutina

¿Qué aspecto tendría una rutina de ensamblador para la obtención de una interrupción?

```
.Model TPascal
.data
int_seg dw ?
int_ofs dw ?

.code
```

```
get_interrupt proc pascal int_nr : BYTE
  mov ah, 35h
  mov al, int_nr
  int 21h
  mov int_ofs, bx
  mov bx, es
  mov int_seg, es
  ret
get_interrupt endp
```

Es algo más complicado si no sobrescribe completamente la interrupción antigua, sino que primero ejecuta su propia rutina y después quiere llamar la interrupción original. En este caso primero ha de averiguar la dirección de la interrupción original y guardarla en una variable *Dword* (*Pointer*). Ahora puede desviar la interrupción a su propia rutina. ¡En esta rutina ha de guardar primero todos los registros que modifique y que son importantes para la rutina destino! Ahora puede procesar su propia rutina. Puede comprobar, por ejemplo, si se ha pulsado una tecla, o si en un lugar determinado de la memoria se encuentra un valor. Después del procesamiento de su código, ha de pensar en restaurar los registros. Ahora salte directamente a la rutina de interrupciones original. No ha de llamar al *iret*, ya que la rutina original lo hará por usted. La estructura podría tener el siguiente aspecto:

```
.data
  oldint dd ?
.code
mi_interrupción proc far
  pusha
  mov ax, cualquier variable
  [...]
  popa
  jmp dword ptr oldint
mi_interrupción endp
```

En todas estas técnicas no podrá prescindir de una solución pura de ensamblador, ya que el Pascal guarda por norma todos los registros cuando se salta a un procedimiento de interrupciones, y, lo que es mucho peor, restaura todos los registros al abandonar este procedimiento. Así no se puede averiguar ningún tipo de valor desde el procedimiento de interrupción. Un último truco, con el que le puede hacer una jugarreta a algunos *Debuggers* o *Junior-Hackers*. Las direcciones de las interrupciones se guardan en la tabla de interrupciones. Esta tabla se encuentra en la memoria principal, en la dirección *$0000: 0000*. Cada entrada mide 4 bytes. Así el valor de la dirección de interrupción se encuentra en la dirección *$0000:Número_interrupción * 4*. Ahora, no siempre ha de emplear las funciones *25h* y *35h* del DOS cuando quiera obtener o modificar la dirección de la interrupción. La puede leer o escribir directamente de/en la tabla en la posición correspondien-

te. En la literatura informática frecuentemente encontrará la indicación de que en ningún caso debe escribir directamente en la tabla, pero también encontrará que debe dibujar un píxel en el modo 13h mediante la interrupción 10h! Así que: ¡olvídelo! No se crean problemas durante la escritura directa de la tabla. De esta forma también puede crear una copia de seguridad de la tabla de interrupciones. Al principio de su programa simplemente copie las 30 entradas en un buffer, restaurándolas al final del mismo. Con ello asegura que todas las interrupciones modificadas vuelven a apuntar a sus rutinas originales.

12.2 El Programmable Interval Timer (PIT)

El PIT es interesante para nosotros en cuanto a la programación de demo siempre que hemos de sincronizar un proceso, o llamarlo periódicamente. Para ello empleamos el contador 0 del PIT. Este controla la interrupción del temporizador del PC, que sino se activa con una frecuencia de 18,2 interrupciones por segundo. Pero antes de mostrarle cómo puede sincronizar a la perfección, por ejemplo una pantalla, vamos a presentarle el hardware que se encuentra detrás del temporizador.

El hardware del PIT

El chip del temporizador es un 8254 PIT. Contiene tres contadores de 16 bits independientes. Cada uno de estos tres contadores puede emplearse según una de las seis siguientes operaciones lógicas:

Modo 0	Generación de interrupciones
Modo 1	Monoflop programable
Modo 2	Generador de reloj
Modo 3	Generador de señales - onda rectangular
Modo 4	Disparo de la salida mediante software
Modo 5	Disparo de la salida mediante hardware

A nosotros nos interesarán especialmente los modos de funcionamiento 1 y 3. Con ellos se puede controlar una adaptación a determinadas frecuencias, una llamada periódica así como eventos sincronizados. Cada uno de estos tres canales es controlado por un contador propio. Este se puede hacer funcionar en modo binario o BCD. El canal 0 controla al temporizador del sistema, que es responsable de operaciones como la hora del sistema, la del día o el Disk timeout. El hardware del sistema utiliza el canal 1. Controla el DRAM-Refresh. El canal 2 finalmente es responsable de la generación de sonido mediante el altavoz del PC.

Un canal se compone de un serie de registros. A nosotros nos interesan sobre todo el registro contador, los dos registros de entrada de contador CIL y CIH de Counter Input Low y Counter Input High, así como el registro de control. Además existen el registro de estado así como dos registros de salida de contador, COL y COH. La siguiente tabla le ofrece un resumen sobre los contadores existentes, las puertas (gates) y señales empleadas:

Contador 0	Gate 0Ch Clk 0Ch Out 0Ch	System Timer genera IRQ0
Contador 1	Gate 1Ch CLK 1Ch OUT 1Ch	DRAM refresh REFTIME
Contador 2	Gate 2Ch CLK 2Ch OUT 2Ch	Frecuencia altavoz Activa altavoz

El registro de control

El registro de control se encarga de la actividad de los diferentes canales. Se puede direccionar mediante el port 43h. Se le pueden pasar el número del canal deseado, el tipo de comando, el modo de trabajo y el formato de contador. Mediante el registro de estado puede leer los ajustes actualmente presentes en el registro de control. Aquí tiene el significado de los bits de este registro:

Bit	Significado
Bit 7-6	Selección de contador
00	Contador 0
01	Contador 1
10	Contador 2
11	Modo de relectura
Bit 5-4	Tipo del comando
00	Latch contador
01	Leer/Escribir el byte bajo del contador
10	Leer/Escribir el byte alto del contador
11	Leer/Escribir primero el byte bajo del contador, después el byte alto
Bit 3-1	Selección del modo de trabajo
000	Modo 0, Generación de interrupción
001	Modo 1, Monoflop programable
010	Modo 2, Generador de reloj
011	Modo 3, Generador de señal - onda cuadrada
100	Modo 4, Disparo de la salida mediante software
101	Modo 5, Disparo de la salida mediante hardware
Bit 0	Elección del formato contador
0	16 Bits Binario (estándar)
1	Formato BCD

El registro del contador

El registro del contador es un registro de 16 bits y cuenta de *0* hasta *0ffffh*. Un acceso de escritura a uno de los registros de entrada del contador, inicia el mismo. Una inicialización del contador *0* tendría el siguiente aspecto:

```
port[43] := $36;
port[40] := 12;
port[40] := 34;
```

Relectura de registro

Los datos de los contadores se pueden volver a leer en los diferentes registros de contador. Para ello se ha de ajustar en el registro de comandos como selección de contador el modo de relectura. Ahora se puede elegir si se quiere leer el valor del contador o el byte de estado. El formato exacto del byte de comando a emplear se puede ver en la siguiente tabla:

Bit	Significado
Bit 7-6	Indicar comando Read Back
11	Ha de tener este valor
Bit 5-4	Selección relectura
01	leer valor actual de contador
10	leer byte de estado
11	no leer nada
Bit 3-0	Elegir contador
1000	Seleccionar contador 2
0100	Seleccionar contador 1
0010	Seleccionar contador 0

El valor del byte de estado leído se ha de interpretar de la siguiente forma:

Bit	Significado
Bit 7	Estado de la señal OUT
0	OUT Signal 0 (low)
1	OUT Signal 1 (high)
Bit 6	¿Activada bandera contadora NIL-COUNT?
0	El contador se fijó mediante el registro de entrada y puede ser leído
1	El registro de control se escribió, pero no los valores de contador esperados
Bit 5-4	Tipo de comando de transferencia
00	Reservado
01	Leer / escribir byte bajo
10	Leer / escribir byte alto
11	Leer / escribir primero byte bajo, después byte alto

Bit	Significado
...	
Bit 3-1	Selección del modo de trabajo
000	Modo 0
001	Modo 1
010	Modo 2
011	Modo 3
100	Modo 4
101	Modo 5
Bit 0	Selección del formato de contador
0	16 bits binario (estándar)
1	Formato BCD

El empleo del PIT

Ahora que ya conoce el hardware del PIT, se preguntará: "¿Estupendo, pero qué tengo yo que hacer con este trasto?". Bueno, a veces ocurre que ha de sincronizar eventos. En la práctica de la programación de demos esto suele ser la llamada periódica de una rutina, por ejemplo para la programación de sonido, o la sincronización con la pantalla.

Normalmente es suficiente con construir la pantalla y esperar el retrazado vertical. Pero a veces es necesario conmutar la pantalla exactamente después del segundo, tercer, etc. retrazado (modo x/y). Puede encontrar un ejemplo de ello en el Fakemode de Yaka y Xography.

¿Cómo puede realizar una conmutación así de forma efectiva? Primero ha de obtener la frecuencia con la que se refresca la pantalla por segundo. En el modo normal del 320 x 200 suele estar en 70 Hz en casi todas las tarjetas. Sólo hemos de generar una interrupción poco antes del final de la representación de una imagen en pantalla, para poder sincronizar con el retrazado y realizar la conmutación. Esto significa para nosotros que, si la interrupción se ha de generar un poco antes, su frecuencia ha de ser ligeramente superior a 70 Hz. Cuánto más depende de la intensidad de cálculo de sus rutinas.

El pequeño programa *Test_timer* trabaja según este principio. Primero obtiene la frecuencia de la construcción pantalla, comprobando mediante la interrupción del temporizador cuanto tiempo puede modificar el valor de una variable durante la espera del retrazado. A continuación se conmuta el temporizador del modo cuadrado al Mono-Flop. La rutina de interrupción ha de programar ahora el temporizador de nuevo cada vez. Para mostrar dónde se realiza la interrupción, la pantalla se conmuta primero al rojo, y después al verde por medio de la interrup-

ción. Así se puede ver muy bien la repartición de los tiempos. A continuación el listado del programa:

```
program test_timer;

uses crt,dos;

Var OTimerInt  : pointer;
    Timerfreq    : word;
    Orig_freq    : word;
    Sync_counter : word;
    Contador_Ti    : word;

PROCEDURE SetColor (Nr, R, G, B : BYTE);
begin;
  asm
    mov   al,Nr
    mov   dx,03C8h
    out   dx,al
    mov   dx,03C9h
    mov   al,r
    out   dx,al
    mov   al,g
    out   dx,al
    mov   al,b
    out   dx,al
  end;
end;

procedure waitretrace;
begin;
  asm
    MOV   DX,03dAh
@WD_R:
    IN    AL,DX
    TEST  AL,8d
    JZ    @WD_R
@WD_D:
    IN    AL,DX
    TEST  AL,8d
    JNZ   @WD_D
  end;
end;

procedure Ajusta_Temporizador(Proc : pointer; Freq : word);
var Contador_I : word;
    oldv : pointer;
begin;
 asm cli end;
```

```
 Contador_I := 1193180 DIV Freq;
 Port[$43] := $36;
 Port[$40] := Lo(Contador_I);
 Port[$40] := Hi(Contador_I);

 Getintvec(8,OTimerInt);
 SetIntVec(8,Proc);
 asm sti end;
end;

procedure Nueva_TimerFreq(Freq : word);
var Contador_I : word;
begin;
 asm cli end;
 Contador_I := 1193180 DIV Freq;
 Port[$43] := $36;
 Port[$40] := Lo(Contador_I);
 Port[$40] := Hi(Contador_I);
 asm sti end;
end;

procedure Apaga_Temporizador;
var oldv : pointer;
begin;
  asm cli end;
  port[$43] := $36;
  Port[$40] := 0;
  Port[$40] := 0;
  SetIntVec(8,OTimerInt);
  asm sti end;
end;

procedure Syncro_interrupt; interrupt;
begin;
  inc(Sync_counter);
  port[$20] := $20;
end;

procedure Syncronize_timer;
begin;
  Timerfreq := 120;
  Ajusta_Temporizador(@Syncro_interrupt,Timerfreq);
  Repeat
    dec(Timerfreq,2);
    waitretrace;
    Nueva_TimerFreq(Timerfreq);
    Sync_counter := 0;
    waitretrace;
  until (Sync_counter = 0);
```

```
end;

Procedure Timer_Handling;
begin;
  setcolor(0,0,63,0);
end;

Procedure Timer_Proc; interrupt;
begin;
  Timer_Handling;
  waitretrace;
  Port[$43] := $34;                    { Modo Mono - Flop }
  Port[$40] := Lo(Contador_Ti);
  Port[$40] := Hi(Contador_Ti);

  setcolor(0,63,0,0);

  port[$20] := $20;
end;

Procedure Inicia_SyncroTimer(Proc : pointer);
var calcl : longint;
begin;
 asm cli end;
  port[$43] := $36;
  Port[$40] := 0;
  Port[$40] := 0;

  Contador_Ti := 1193180 DIV (Timerfreq+5);
  setintvec(8,Proc);
  waitretrace;
  Port[$43] := $34;                    { Modo Mono - Flop }
  Port[$40] := Lo(Contador_Ti);
  Port[$40] := Hi(Contador_Ti);
 asm sti end;
end;

begin;
  clrscr;
  Syncronize_Timer;
  writeln('La frecuencia de temporizador es : ',Timerfreq);
  Inicia_SyncroTimer(@Timer_Proc);
  repeat until keypressed;
  while keypressed do readkey;
  Apaga_Temporizador;
  setcolor(0,0,0,0);
end.
```

12.3 El Programmable Interrupt Controller (PIC)

También el PIC es un elemento importante para nosotros. Controla las interrupciones del PC. Exactamente, el PC dispone de dos controladores de interrupciones. El primero se encuentra en la dirección *20h*, el segundo en *A0h*. Pero el segundo controlador está unido al primero mediante una conexión en cascada. Esta se realiza a través del canal 2 del controlador en *20h*.

El PIC trabaja según la lógica de prioridades. Es decir, la interrupción con el valor menor, la interrupción *0*, posee la prioridad mayor. La prioridad inferior la posee por ello la interrupción *7* y no, como se podría suponer, la interrupción *15*. Esto es porque las interrupciones *8* hasta *15* se incluyen mediante la interrupción 2, y en la lógica de preferencias están colocadas por delante de la interrupción *3*. La siguiente tabla ofrece un resumen de esta lógica de prioridades:

Prio.	PIC Nº	Int.-Nº	Significado
1	-	NMI	Todas las interrupciones no enmascarables
2	1	IRQ 0	Interval Timer
3	1	IRQ 1	Interrupción de teclado
4	2	IRQ 8	RTC
5	2	IRQ 9	IRQ 2 emulada por software
6	2	IRQ 10	No empleado
7	2	IRQ 11	No empleado
8	2	IRQ 12	Ratón u otro dispositivo de entrada
9	2	IRQ 13	Interrupción de coprocesador
10	2	IRQ 14	Controlador de disco duro
11	2	IRQ 15	No empleado
12	1	IRQ 3	2º puerto serie
13	1	IRQ 4	1º puerto serie
14	1	IRQ 5	2º puerto paralelo
15	1	IRQ 6	Controlador de disquetes
16	1	IRQ 7	1º puerto paralelo

A causa de esta lógica de prioridades el PIC simplifica el manejo del sistema operativo controlador por interrupciones y ayuda a reducir el gasto de gestión.

Aquí no hablaremos de los detalles del PIC. Quien quiera informaciones más extensas, las encontrará en la literatura correspondiente sobre la programación del sistema (por ejemplo PC Interno 2.0, de MARCOMBO, S.A.). En este lugar sólo queremos hablar del empleo en el día a día de los programadores. El PC distingue entre cuatro tipos de interrupciones:

- Interrupciones enmascarables (MI)
- Interrupciones no enmascarables (NMI)

- Interrupciones de hardware
- Interrupciones de software

Interrupciones enmascarables

Las interrupciones enmascarables son disparadas por el hardware del PC. Se pueden evitar mediante el empleo de la instrucción de ensamblador

```
cli     ;Borra todas las interrupciones enmascarables.
```

Como parte opuesta, con

```
sti     ;Activar interrupciones de nuevo
```

puede volverlas a activar. Además existe la posibilidad de apagar las interrupciones de forma selectiva. Esto ocurre enmascarándolas en el port correspondiente. Para las interrupciones *IRQ 0* hasta *IRQ 7* esto depende el port *020h*, para las demás, el port *0A0h*.

Interrupciones no enmascarables

Algo más complicado es el tema de las interrupciones no enmascarables. Son llamadas cuando aparece, por ejemplo, un error de paridad, o cuando el BIOS dispara esta interrupción. También estas interrupciones se pueden apagar. Esto nos puede interesar, por ejemplo, cuando se realizan operaciones muy críticas (de tiempo), y no se quiere que sean interrumpidas. Mediante la instrucción de ensamblador

```
mov al, 80h
out 70h, al
```

puede apagar las NMI. Con

```
mov al, 00h
out 70h, al
```

se vuelven a activar.

Interrupciones de hardware

Bajo estas se entienden todas las interrupciones que son disparadas por hardware instalado adicionalmente. Generan la mayoría de interrupciones en uno de los canales de interrupción libres. Su tratamiento es equivalente al de las interrupciones enmascarables.

Interrupciones de software

Las interrupciones de software son todas aquellas, que son disparadas por el software. Esto pueden ser interrupciones de sistema del sistema operativo, como por

ejemplo la *interrupción 21h*. Pero también se puede tratar de interrupciones instaladas por el usuario para controlar hardware o para un tratamiento genérico. Se pueden apagar mediante la instrucción *cli* antes mencionada y reactivar mediante *sti*.

12.4 El controlador DMA

Mediante el controlador DMA se pueden transferir rápidamente grandes bloques de datos de un dispositivo periférico a la memoria o al revés. La técnica empleada es algo radical: el controlador DMA simplemente desconecta el procesador y transfiere entonces los datos.

El PC dispone de dos controladores DMA. El primer controlador es responsable de las transferencias de 8 bits y el segundo regula las de 16 bits. El segundo controlador dispone de una conexión en cascada al primero mediante el canal *0*. Aquí tiene un resumen de los diferentes canales:

Canal	Controlador	Descripción
Canal 0	Controlador 1	RAM-refresh
Canal 1	Controlador 1	libre, habitualmente SB, GUS etc.
Canal 2	Controlador 1	Unidad de disquetes
Canal 3	Controlador 1	Unidad de disco duro
Canal 4	Controlador 2	Cascada Contr.1 => Contr. 2
Canal 5	Controlador 2	libre
Canal 6	Controlador 2	libre
Canal 7	Controlador 2	libre

Para nosotros, en la práctica, es interesante lo siguiente: ¿Cómo se ha de programar el controlador DMA, para que por ejemplo se le puedan enviar datos a una Sound Blaster? El modo de proceder es el siguiente:

1. Bloquear canal DMA
2. Ajustar modo de transferencia
3. Borrar Flip-Flop
4. Escribir dirección del bloque de datos
5. Escribir página del bloque de datos
6. Escribir longitud de transferencia
7. Liberar canal DMA de nuevo

El enmascarado del canal DMA

Ahora nos hemos de ocupar de cómo podemos convertir a la práctica los diferentes pasos de este modo de proceder. Ocupémonos primero del (des)enmascarado de un canal DMA. Para enmascarar un canal DMA hay dos posibilidades:

1. El enmascarado mediante el *Single-Mask-Register*

Mediante este registro se puede (des)enmascarar un solo canal. El registro se encuentra en el port *0Ah* para los canales *0 - 3* y en el port *0D4h* para los canales *4 - 7*. Los bits tienen el siguiente significado:

Bits	Significado
Bit 7-3	no empleado
Bit 2	Fijar o borrar la máscara
1	Fijar la máscara
0	Borrar la máscara
Bit 1-0	Selección del canal
00	Canal 0 del controlador correspondiente
01	Canal 1 del controlador correspondiente
10	Canal 2 del controlador correspondiente
11	Canal 3 del controlador correspondiente

Esto significa que cuando quiere enmascarar el canal *1* del controlador *1* (lo que necesita para la Sound Blaster estándar), ha de escribir el valor *5* en el port *0Ah*. Para volver a activar el canal después de la programación del controlador DMA, escriba el valor *1*.

2. El enmascarado mediante el *All-Mask-Register*

Mediante el *All-Mask-Register* puede (des)enmascarar todos los canales DMA de un controlador DMA de una sola vez. La ventaja es que no ha de enmascarar cada canal como en el caso del *Single-Mask-Register*. Sin embargo ha de indicar para cada canal si se ha de activar o desactivar. Así que si no sabe lo que ocurre en todos los canales debería elegir preferentemente las variante *Single-Mask*. El *All-Mask-Register* se encuentra en el port *0Fh* para el primer controlador y en el port *0DEh* para el segundo. Un bit de máscara (*Mask-bit*) activo indica que no se pueda acceder al canal, un bit borrado permite el acceso. Aquí tiene la ocupación del byte a escribir:

Bits	Significado
Bit 7-4	No empleado
Bit 3	(Des)enmascarar el canal 3
1	Activar máscara del canal 3
0	Desactivar máscara del canal 3
Bit 2	(Des)enmascarar el canal 2
1	Activar máscara del canal 2
0	Desactivar máscara del canal 2
Bit 1	(Des)enmascarar el canal 1
1	Activar máscara del canal 1
0	Desactivar máscara del canal 1
Bit 0	(Des)enmascarar el canal 0
1	Activar máscara del canal 0
0	Desactivar máscara del canal 3

El ajuste del modo de transferencia

El controlador DMA domina varios modos de transferencia. Al principio de cada transferencia DMA ha de ajustar el modo necesitado. Para ello ha de enviar el bit de modo al port *0Bh* para el primer controlador DMA, y al port *0D6h* para el segundo. Primero un resumen de la ocupación del byte:

Bit	Significado
Bit 7-6	Selección de modo
00	Modo de petición
01	Modo singular
10	Modo bloque
11	Modo cascada
Bit 5	Incrementar / decrementar direcciones
1	Dirección de decremento
0	Dirección de incremento
Bit 4	Autoinicialización si/no
1	Activar autoinicialización
0	Desactivar autoinicialización
Bit 3-2	Selección de transferencia
00	Transferencia Verify
01	Transferencia de escritura
10	Transferencia de lectura
11	no permitido
Bit 1-0	Selección de canal
00	Emplear canal 0
01	Emplear canal 1
10	Emplear canal 2
11	Emplear canal 3

Veamos un poco más de cerca el significado de los diferentes bits:

Selección de modo

Mediante los dos bits 7 - 6 se elige el modo a emplear durante la transferencia. Se ha de distinguir entre cuatro modos:

- Modo de petición
- Modo singular
- Modo bloque
- Modo cascada

El modo interesante para nosotros es el modo singular. En este, siempre se transfiere un byte, y a continuación se incrementa o decrementa el contador de direcciones, según el modo programado. Cuando el contador alcance su valor destino, se realiza (si se ha ajustado así), una autoinicialización.

Incrementar/decrementar direcciones

Mediante el bit 5 puede determinar si el bloque elegido se ha de transferir hacia adelante (incremento) o hacia atrás (decremento).

Autoinicialización si/no

Con el bit 4 determina si al final de la transferencia se realiza una autoinicialización o no. Esta provoca que el registro de direcciones y el registro de contador se repongan a los valores programados. Mediante la activación de este bit se puede realizar un bonito truco: si programa un MOD-Player para la tarjeta Sound Blaster, puede trabajar con un Double-Buffer o con un buffer en anillo. Si elige este último, puede indicarle a la SB que está transfiriendo un bloque muy grande (*hasta 0FFFFh*). Sin embargo, programe el controlador DMA al tamaño real del bloque. Esto provoca que el bloque se vuelve a reproducir desde el principio cuando finaliza (principio de buffer en anillo). De esta forma se puede ahorrar una cantidad sustancial de llamadas a la rutina de inicialización DMA y la interrupción de la SB se llamará con menos frecuencia. Esto trae consigo aumentos de la velocidad y reduce los posibles chasquidos.

Selección de transferencia

Mediante los dos bits 3 y 2 puede ajustar la dirección de la transferencia a realizar. Una transferencia de escritura significa una transferencia desde la fuente seleccionada hasta la memoria principal. Mediante una transferencia de lectura pasa datos de la memoria al destino elegido. Una transferencia Verify comprueba además la transferencia realizada.

Selección de canal

Aquí ha de ajustar qué canal se ha de emplea para la operación de transferencia. Tenga en cuenta que el canal DMA *4* corresponde al canal *0* del segundo controlador. Las diferencias entre el primer y segundo controlador se encuentran básicamente en las direcciones de port distintas.

El borrado del Flip-Flop DMA

Antes de algunas acciones de programación del controlador DMA es necesario borra el Flip-Flop. Esto lo puede realizar mediante el registro *Clear-Flipflop*. Se encuentra en el port *0Ch* para el primer controlador y en *0D8h* para el segundo. Puede borrar el registro, escribiendo el valor *0* en el port correspondiente.

Ajustar la dirección del bloque de datos

El controlador DMA necesita la página física del bloque de datos a transferir. Esta se calcula según la fórmula:

```
direcc := 16 * longint(Seg(Bloque^))+ofs(Bloque^);
```

Los 16 bits superiores del número de 32 bits así obtenido indican la página, y los 16 bits bajos el offset. Para fijar la dirección, se ha de escribir primero el offset en el registro de direcciones DMA. Según el canal se encuentra en los siguientes ports:

Canal 0	Port 00h
Canal 1	Port 02h
Canal 2	Port 04h
Canal 3	Port 06h
Canal 4	Port C0h
Canal 5	Port C4h
Canal 6	Port C8h
Canal 7	Port CCh

En el port correspondiente se ha de escribir primero el byte bajo y a continuación el byte alto del offset de la dirección. Ahora puede transferir la página al controlador. Para ello existen dos registro diferentes, un registro *Lower-Page* para los 8 bits bajos de la dirección y un registro *Upper-Page* para los 4 bits superiores de la misma. Así se pueden direccionar hasta 256 MBytes de memoria mediante las páginas, lo que debería suficiente en cualquier caso. El registro *Lower-Page* sobrescribe el contenido del registro *Upper-Page* con 0, cuando es programado. Por ello ha de ser escrito antes del registro *Upper-Page*. La siguiente tabla ofrece un resumen de los ports de estos registros:

Canal	Lower Page	Upper Page
0	87h	487h
1	83h	483h
2	81h	481h
3	82h	482h
4	0	0
5	8Bh	48Bh
6	89h	489h
7	8Ah	48Ah

Ajustar el tamaño de una transferencia DMA

Antes de poder escribir el tamaño del bloque a transferir en el controlador DMA, ha de borrar primero el Flip-Flop. Después se transfiere el byte bajo y a continuación el byte alto al registro *DMA-Count*. Con ello se ha determinado el número de datos a transferir. Aquí están los ports del registro contador.

Canal 0	Port 01
Canal 1	Port 03
Canal 2	Port 05
Canal 3	Port 07h
Canal 4	Port C0h
Canal 5	Port C4h
Canal 6	Port C8h
Canal 7	Port CCh

A continuación una pequeña Unit para controlar los accesos DMA. Pone a su disposición todos los procedimientos necesitados. Puede programar todos los pasos de forma individual o realizar una inicialización completa mediante el procedimiento *DMA_Init_Transfer*. Como primer parámetro se le ha de pasar al procedimiento el número del canal a emplear, después el modo deseado y un puntero al bloque de datos en la memoria principal y finalmente el tamaño del bloque de datos.

```
unit DMA;

interface

TYPE DMAarray = array[0..7] of byte;

CONST
 { direcciones del controlador DMA }
 DMA_Adress              : DMAarray =($00,$02,$04,$06,$C0,$C4,$C8,$CC);
 DMA_Count               : DMAarray =($01,$03,$05,$07,$C2,$C6,$CA,$CE);
```

```
DMARead_status_Reg     : DMAarray =($08,$08,$08,$08,$D0,$D0,$D0,$D0);
DMAWrite_status_Reg    : DMAarray =($08,$08,$08,$08,$D0,$D0,$D0,$D0);
DMAWrite_requ_Reg      : DMAarray =($09,$09,$09,$09,$D2,$D2,$D2,$D2);
DMAWr_single_mask_Reg : DMAarray =($0A,$0A,$0A,$0A,$D4,$D4,$D4,$D4);
DMAWr_mode_Reg         : DMAarray =($0B,$0B,$0B,$0B,$D6,$D6,$D6,$D6);
DMAClear_Flipflop      : DMAarray =($0C,$0C,$0C,$0C,$D8,$D8,$D8,$D8);
DMARead_Temp_Reg       : DMAarray =($0D,$0D,$0D,$0D,$DA,$DA,$DA,$DA);
DMAMaster_Clear        : DMAarray =($0D,$0D,$0D,$0D,$DA,$DA,$DA,$DA);
DMA_Clear_Mask_Reg     : DMAarray =($0E,$0E,$0E,$0E,$DC,$DC,$DC,$DC);
DMA_Wr_All_Mask_Reg    : DMAarray =($0F,$0F,$0F,$0F,$DE,$DE,$DE,$DE);
DMA_Lower_Page         : DMAarray =($87,$83,$81,$82,$00,$8B,$89,$8A);
DMA_Higher_Page        : Array[0..7] of word
                           = ($487,$483,$481,$482,$0,$48B,$489,$48A);

{ registro de modo DMA_Wr_mode_Reg }
ModoPeticion  = $00;
ModoSingular  = $40;
ModoBloque    = $80;
ModoCascada = $C0;
Decremento_Direccion  = $20;
Incremento_Direccion  = $00;
Autoinit_Enable       = $10;
Autoinit_Disable      = $00;
Comprobar_Trans       = $00;
Escribir_Trans        = $04;
Leer_Trans            = $08;

Set_Request_Bit       = $04;
Clear_Request_Bit     = $00;
Set_Mask_Bit          = $04;
Clear_Mask_Bit        = $00;
procedure DMA_Estab_Modo(Canal,Modo : byte);
procedure DMA_Estab_ModoNorm(Canal,Modo : byte);
procedure DMA_Clear_Flipflop(Canal : byte);
procedure DMA_DirecIni(Canal : byte; Start : pointer);
procedure DMA_TamanyoBloque(Canal : byte; size : word);
procedure DMA_Enmascarar_Canal(Canal : byte);
procedure DMA_Desenmascarar_Canal(Canal : byte);
procedure DMA_Init_Transfer(Canal,Modo :byte; p :pointer; s :word);

implementation

TYPE
  pt = record                          { permite el tratamiento   }
    ofs,sgm  :  word;                  { sencillo de punteros     }
  end;

procedure DMA_Estab_Modo(Canal,Modo : byte);
```

```
begin;
  port[DMAWr_Mode_Reg[Canal]] := Modo;
end;

procedure DMA_Estab_ModoNorm(Canal,Modo : byte);
begin;
  port[DMAWr_Mode_Reg[Canal]] :=Modo+Incremento_Direccion+Leer_Trans+
                                Autoinit_Disable+Canal;
end;

procedure DMA_Clear_Flipflop(Canal : byte);
begin;
  port[DMAClear_Flipflop[Canal]] := 0;
end;

procedure DMA_DirecIni(Canal : byte; Start : pointer);
var l : longint;
    pn,offs : word;
begin;
  l := 16*longint(pt(Start).sgm)+pt(Start).ofs;
  pn := pt(l).sgm;
  offs := pt(l).ofs;
  port[DMA_Adress[Canal]] := lo(offs);
  port[DMA_Adress[Canal]] := hi(offs);
  port[DMA_Lower_Page[Canal]] := lo(pn);
  port[DMA_Higher_Page[Canal]] := hi(pn);
end;

procedure DMA_TamanyoBloque(Canal : byte; size : word);
begin;
  DMA_Clear_Flipflop(Canal);
  port[DMA_Count[Canal]]:= lo(size);
  port[DMA_Count[Canal]] := hi(size);
end;

procedure DMA_Enmascarar_Canal(Canal : byte);
begin;
 port[DMAWr_single_mask_Reg[Canal]] := Canal + Set_Mask_Bit;
end;

procedure DMA_Desenmascarar_Canal(Canal : byte);
begin;
 port[DMAWr_single_mask_Reg[Canal]] := Canal + Clear_Mask_Bit;
end;

procedure DMA_Init_Transfer(Canal,Modo :byte; p :pointer; s :word);
begin;
  DMA_Enmascarar_Canal(Canal);
  DMA_DirecIni(Canal,p);
```

```
    DMA_TamanyoBloque(Canal,s);
    DMA_NormModo_Fijar(Canal,Modo+Canal);
    DMA_Desenmascarar_Canal(Canal);
  end;

begin;
end.
```

12.5 El reloj de tiempo real (RTC)

Un chip muy importante es el reloj de tiempo real (Real-Time-Clock, RTC). Se trata de un *Dallas 1287* o similar. Está mantenido por baterías y trabaja en un modo de baja energía que lo protege de pérdidas de datos durante la conexión o desconexión del equipo.

En la RAM RTC de 64 bytes, a veces también llamada CMOS-RAM, se guarda la configuración del sistema. Además, el chip nos ofrece un reloj, un calendario, así como la posibilidad de una interrupción periódica programable.

La actualización del tiempo y el calendario se realiza de forma cíclica. El contador de segundo se incrementa en uno cada segundo, siempre que el RTC se encuentre en el modo de funcionamiento normal. Si se realiza un rebase, los demás registros afectados también se incrementan.

Durante este rebase, los bytes *0* hasta *9* de la RAM RTC no están disponibles para la CPU. La velocidad del ciclo de actualización se ajusta mediante los bits divisores 2 hasta *0* del registro de estado *A*, así como el SET-Bit *7* del registro de estado *B*.

El acceso a la RAM RTC es muy sencillo. Primero se ha de enviar al port *70h* el número de índice del registro deseado. Después se puede escribir o leer los datos en el port *71h*. Todos los registros son de lectura y escritura, excepto los siguientes que sólo se pueden leer:

```
Registro de estado C y D
Bit 7 del registro de estado A
Bit 7 del byte de segundos (Index 00h)
```

Los primeros 14 bytes de la RAM RTC se emplean para el reloj y los cuatro registros de estado. Los restantes 50 bytes sirven para la configuración del sistema. La ocupación exacta de los diferentes bytes se puede ver en la siguiente tabla:

00h	Segundos	20h	HD 1 Número cilindros Low
01h	Segundos Alarma	21h	HD 1 Número cilindros High
02h	Minutos	22h	Cabezales HD 1
03h	Minutos Alarma	23h	HD 1 Inicio Pre-Compensation Low
04h	Horas	24h	HD 1 Inicio Pre-Compensation High
05h	Horas Alarma	25h	HD 1 Low Landing zone
06h	Día de la semana	26h	HD 1 High Landing zone
07h	Día del mes	27h	HD 1 Sectores
08h	Mes	28h	Opciones 1
09h	Año	29h	Reservado
0Ah	Estado A	2Ah	Reservado
0Bh	Estado B	2Bh	Opciones 2
0Ch	Estado C	2Ch	Opciones 3
0Dh	Estado D	2Dh	Reservado
0Eh	Estado de diagnóstico	2Eh	Low CMOS Ram Checksum
0Fh	Estado Shutdown	2Fh	High CMOS Ram Checksum
10h	Tipo de Floppy	30h	Low Extended Memory byte
11h	Reservado	31h	High Extended Memory byte
12h	Tipo de HD	32h	byte de siglo
13h	Reservado	33h	Setup Information
14h	Equipamiento	34h	Velocidad CPU
15h	Low Base Memory	35h	HD 2 Número de cilindros Low
16h	High Base Memory	36h	HD 2 Número de cilindros High
17h	Low Extended Memory	37h	Cabezales HD 2
18h	High Extended Memory	38h	HD 2 Inicio Pre-Compensation Low
19h	HD 1 extended Type byte	39h	HD 2 Inicio Pre-Compensation High
1Ah	HD 2 extended Type byte	3Ah	HD 2 Low Landing zone
1Bh	Reservado	3Bh	HD 2 High Landing zone
1Ch	Reservado	3Ch	HD 2 Sectores
1Dh	Reservado	3Dh	Reservado
1Eh	Reservado	3Eh	Reservado
1Fh	Installed Features	3Fh	Reservado

Las funciones de reloj

La CPU obtiene la hora y la fecha leyendo el byte correspondiente del RTC. Si este se está incrementando actualmente, la lectura no es posible. Mediante la escritura de un valor en los bytes, estos se inicializan.

Si quiere escribir datos en los bytes, ha de apagar primero los *RTC-Updates* en el registro de estado *B* mediante el *SET*-Bit. Los datos del reloj están guardados en formato BCD. Es decir, que los 4 bits superiores representan el número de las decenas y los 4 bits inferiores las unidades de un número. Los siguientes valores se pueden encontrar en los diferentes bytes:

Función	Dirección	Datos BCD
Segundos	00	00 hasta 59
Segundos Alarma	01	00 hasta 59
Minutos	02	00 hasta 59
Minutos Alarma	03	00 hasta 59
Día de la semana	06	01 hasta 07
Día del mes	07	01 hasta 31
Mes	08	01 hasta 12
Año	09	00 hasta 99

Los registros de estado

El RTC dispone de cuatro registros de estado. Sirven para el control del chip y muestran su estado actual.

Registro A

El registro *A* es especialmente interesante por la posibilidad de determinar con él la velocidad de la llamada de la interrupción periódica. En el bit *7* puede encontrar el bit *UIP* (=Update in Progress). Si tiene el valor *1*, se está realizando una actualización del tiempo en este instante. Los bits 6 hasta 4 contienen la base de tiempo. El valor estándar para ella es de *101b*, lo que representa un valor de *32768d Hz*. Debería prestar especial atención a los bits *RS 3* hasta *0*. Son responsables de la selección de tasas. La tasa se calcula según la fórmula

```
Tasa = 65536 / 2^RS.
```

El valor por defecto de 1024 Hz se calcula por consiguiente de 65536 / 2^6 (1001b).

Registro B

El registro B es un registro con muchas funciones. Cada bit tiene la suya propia. Comencemos con el *SET*-Bit *7*. Si está a *1*, se interrumpe el ciclo momentáneo de actualización. Esto se necesita sobre todo para la inicialización. Con un valor de *0* se realiza una actualización una vez por segundo. El *PIE*-Bit *6* (Periodic Interrupt Enable) determina si la interrupción periódica se ha de llamar con la frecuencia ajustada en el registro *A*. Si el bit está a *1*, se realiza la llamada, de lo contrario no se dispara ninguna interrupción. En el *AIE*-Bit *5* (Alarm Interrupt Enable) puede encontrar la información sobre si la interrupción de alarma se disparará a la hora ajustada.

Si el bit vale *1*, la interrupción está activa. La interrupción de actualización se dispara cuando se puede encontrar el valor *1* en el *UIE*-Bit *4* (Update Ended Interrupt Enable). Con el *SQWE*-Bit *3* (Square Wave Enable) puede elegir si se debe activar la

frecuencia de onda rectangular ajustada en el registro *A* (Bit = 1) o no. El *DM*-Bit 2 (Date Mode) es importante cuando accede a la fecha. Si el bit contiene el valor *1*, los números se encuentran en formato binario.

Por defecto puede encontrar aquí el valor *0*, que indica el formato BCD. Mediante el *24/12*-Bit *1* (24/12 hour) se elige si el reloj se encuentra en el modo de 24 horas o en el de 12 horas. Por defecto puede encontrar aquí el valor *1*, que representa el modo de 24 horas. En el *DSE*-Bit *0* (Daylight Savings Enabled) puede finalmente encontrar la información de si el reloj ha de tener en cuenta el horario de verano o no. De forma estándar esto está apagado mediante un valor de *0*.

Registro C

El registro *0Ch* es el registro de banderas. El bit 7 indica el *Interrupt-Request*-Flag. Contiene el valor *1* cuando se puede aplicar uno de los estados que disparan una interrupción y la bandera correspondiente está en *enable*. El bit 6 es una bandera que indica una interrupción periódica. El bit 5 se pone a *1* cuando hay una coincidencia entre la hora de alarma ajustada y la hora actual. La bandera también se activa cuando la interrupción de alarma no está activada. El bit 4 finalmente es la llamada *Update Ended Interrupt-Flag*. Indica, como se nombre permite suponer, cuando se ha terminado un ciclo de actualización del RTC. Los bits 3 hasta 0 están reservado. Por favor tenga en cuenta que el contenido de este registro es borrado después de cada acceso de lectura.

Registro D

El registro *D* sirva para la vigilancia de la batería. Si el bit 7 está activado, la batería se encuentra en perfecto estado. De lo contrario, un valor de *0* muestra que la batería está defectuosa. Los bits 6 hasta 0 no tienen ninguna función.

Los bytes de configuración

En los bytes a partir de *0Eh* se encuentra la configuración del sistema. A continuación vamos a presentarle brevemente la ocupación de los distintos bytes. Puede encontrar informaciones más extensas en la literatura sobre la programación del sistema como *PC Interno* 2.0 de MARCOMBO, S.A.

El byte de diagnóstico de estado 0Eh

Este byte sirve para la comprobación de la configuración correcta del sistema al inicializarlo. Los diferentes bits significan:

Bit	Significado
Bit 7	Estado de la batería RTC
1	No hay tensión
0	Hay tensión
Bit 6	Control de suma de control
1	Suma de control no válida
0	Suma de control válida
Bit 5	¿Informaciones correctas de configuración?
1	Configuración no válida
0	Configuración válida
Bit 4	Comprobación de tamaño de memoria modificado
1	Tamaño de memoria se ha modificado
0	Tamaño de memoria no se ha modificado
Bit 3	Comprobación de disco duro
1	Error de disco duro o controladora
0	Sin errores
Bit 2	Estado de la hora
1	Hora incorrecta
0	Hora correcta
Bit 1-0	Reservado

El Shutdown-Status-byte 0Fh

Este byte se activa siempre en un reset de la CPU. El código de reset sirve como indicador del tipo de reset. Son posibles las siguientes ocupaciones:

00h	Reset normal del sistema
09h	Software-Reset (retorno del Protected Mode)

Tipo de Floppys conectados - 10h

Mediante este byte se pueden obtener todas las informaciones acerca de las unidades de disquete conectadas.

Bit	Significado
Bit 7-4	Tipo de la primera unidad de disquete
0000	No hay unidad instalada
0001	Unidad de 360 KBytes, 5.25"
0010	Unidad de 1.2 MBytes, 5.25"
0011	Unidad de 720 KBytes, 3.5"
0100	Unidad de 1.44 MBytes, 3.5"
0101	Unidad de 2.88 MBytes, 3.5"
...	

Bit	Significado
...	
Bit 3-0	Tipo de la segunda unidad de disquetes
0000	No hay unidad instalada
0001	Unidad de 360 KBytes, 5.25"
0010	Unidad de 1.2 MBytes, 5.25"
0011	Unidad de 720 KBytes, 3.5"
0100	Unidad de 1.44 MBytes, 3.5"
0101	Unidad de 2.88 MBytes, 3.5"

Tipo de los discos duros instalados - 12h

El tipo de los discos duros instalados se puede determinar mediante este byte, aunque habitualmente necesita los bytes *19h* y *1Ah* para informaciones adicionales. El byte tiene el siguiente significado:

Bit	Significado
Bit 7-4	Tipo del primer disco duro
0000	No hay disco duro instalado
0001-1110	Disco del tipo 1 hasta 14
1111	Información de tipo en el byte 19h
Bit 3-0	Tipo del segundo disco duro
0000	No hay disco duro instalado
0001-1110	Disco del tipo 1 hasta 14
1111	Información de tipo en el byte 1Ah

Definición del equipamiento - 14h

Este byte se emplea para el auto-test de hardware. Aquí tiene su significado:

Bit	Significado
Bit 7-6	Número de Floppys instalados
00	Una unidad de disquetes
01	Dos unidades de disquetes
10	Reservado
11	Reservado
Bit 5-4	Tipo de tarjeta gráfica
00	Extended functionality controller
01	Pantalla color con 40 columnas
10	Pantalla color con 80 columnas
11	Pantalla monocroma
Bit 3-2	No empleado
...	

Bit	Significado
...	
Bit 1	Coprocesador
1	Coprocesador está instalado
0	Coprocesador no está instalado
Bit 0	¿Unidad de disquetes instalada?
1	Unidad instalada
0	No hay unidad

Low/High Base Memory - 15h/16h

De los bytes Low-Base-Memory y High-Base-Memory se puede averiguar el tamaño de la memoria principal instalada. El valor *0200h* por ejemplo representa 512 KBytes de RAM, *0280h* corresponde a 640 KBytes.

Low/High Extended Memory - 17h/18h

De los bytes de Low- y High-Extended-Memory se puede averiguar el tamaño de la memoria extendida (mayor de 1 MByte) instalada. Así, por ejemplo, 0400h representa a 1024 Kbytes de extended RAM y 0C00 a 3072 KBytes.

Extended-Type-byte para disco duro 1- 19h

En este byte se encuentra el tipo del primer disco duro, cuando los bits 7 hasta 4 del byte *12h* tienen el valor *0Fh*.

Extended-Type-byte para disco duro 2- 1Ah

En este byte se encuentra el tipo del primer disco duro, cuando los bits 3 hasta 0 del byte *12h* tienen el valor *0Fh*.

Installed Features- 1Fh

Este byte se necesita para las funciones de mensajes de error.

Bit 7-3	Reservado
Bit 2	Avisar error de disquetera
Bit 1	Avisar error de pantalla
Bit 0	Avisar error de teclado

byte de velocidad de CPU - 34h

Mediante este byte se puede determinar la velocidad de la CPU. Los bits 7 hasta 1 están reservados. Si el bit 0 tiene el valor *0h*, la CPU se encuentra en su modo lento, con el bit activado trabaja en modo Turbo.

El RTC en la práctica

El programa que presentamos a continuación, demuestra el acceso al RTC y la evaluación de los registros. Puede servir como base para una Unit de control del RTC.

```
program rtc_unit;

uses crt,dos;

const
  Rtc_Segundos            = $00;
  Rtc_Segundos_alarm      = $01;
  Rtc_Minutos             = $02;
  Rtc_Minutos_alarm       = $03;
  Rtc_Horas               = $04;
  Rtc_Horas_alarm         = $05;
  Rtc_dia_semana          = $06;
  Rtc_dia_mes             = $07;
  Rtc_Mes                 = $08;
  Rtc_Anyo                = $09;
  Rtc_Status_A            = $0A;
  Rtc_Status_B            = $0B;
  Rtc_Status_C            = $0C;
  Rtc_Status_D            = $0D;
  Rtc_estado_diagnostico  = $0E;
  Rtc_Shutdown_status     = $0F;
  Rtc_Floppy_Typ          = $10;
  Rtc_HD_Typ              = $12;
  Rtc_Equipamiento        = $14;
  Rtc_Lo_Basememory       = $15;
  Rtc_Hi_Basememory       = $16;
  Rtc_Lo_Extendedmem      = $17;
  Rtc_Hi_Extendedmem      = $18;
  Rtc_HD1_extended        = $19;
  Rtc_HD2_extended        = $1A;
  Rtc_Features            = $1F;
  Rtc_HD1_Lo_Cylinder     = $20;
  Rtc_HD1_Hi_Cylinder     = $21;
  Rtc_HD1_Cabezales       = $22;
  Rtc_HD1_Lo_Precom       = $23;
  Rtc_HD1_Hi_Precom       = $24;
  Rtc_HD1_Lo_Landing      = $25;
  Rtc_HD1_Hi_Landing      = $26;
  Rtc_HD1_Sectores        = $27;
  Rtc_Opciones1           = $28;
  Rtc_Opciones2           = $2B;
  Rtc_Opciones3           = $2C;
  Rtc_Lo_Checksum         = $2E;
```

```
  Rtc_Hi_Checksum          = $2F;
  Rtc_Extendedmem_Lo       = $30;
  Rtc_Extendedmem_Hi       = $31;
  Rtc_Anyohundert          = $32;
  Rtc_Setup_Info           = $33;
  Rtc_CPU_speed            = $34;
  Rtc_HD2_Lo_Cylinder      = $35;
  Rtc_HD2_Hi_Cylinder      = $36;
  Rtc_HD2_Cabezales        = $37;
  Rtc_HD2_Lo_Precom        = $38;
  Rtc_HD2_Hi_Precom        = $39;
  Rtc_HD2_Lo_Landing       = $3A;
  Rtc_HD2_Hi_Landing       = $3B;
  Rtc_HD2_Sectores         = $3C;


function wrhexb(b : byte) : string;
const hexcar : array[0..15] of char =
 ('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F');
begin;
  wrhexb := hexcar[(b shr 4)] + hexcar[(b AND $0F)];
end;

function wrhexw(w : word) : string;
begin;
  wrhexw := '$'+wrhexb(hi(w))+wrhexb(lo(w));
end;

procedure write_rtc(Reg,val : byte);
{
 Escribe un valor en el registro RTC indicado en RTC
}
begin;
  port[$70] := Reg;
  port[$71] := val;
end;

function read_rtc(Reg : byte) : byte;
{
 lee un valor del registro RTC indicado en Reg
}
begin;
  port[$70] := Reg;
  read_rtc := port[$71];
end;

Procedure Write_Floppy;
{
 visualiza más informaciones sobre las unidades de floppy instaladas
```

```
}
Var Fl : byte;
    Fls : array[1..2] of byte;
begin;
  Fl      := Read_Rtc(Rtc_Floppy_Typ);
  Fls[2] := Fl AND $0F;
  Fls[1] := Fl SHR 4;
  for Fl := 1 to 2 do begin;
    write('Floppy ',Fl,': ');
    case Fls[Fl] of
      0 : begin;
            writeln('No  Floppy         ');
          end;
      1 : begin;
            writeln('5¼" Floppy, 360 KB');
          end;
      2 : begin;
            writeln('5¼" Floppy, 1.2 MB');
          end;
      3 : begin;
            writeln('3½" Floppy, 720 KB');
          end;
      4 : begin;
            writeln('3½" Floppy, 1.44 MB');
          end;
    end;
  end;
end;

Procedure Write_Hd;
{
 Indica el tipo de los HD instalados
}
Var Hd : byte;
    Hds : array[1..2] of byte;
begin;
  Hd      := Read_Rtc(Rtc_HD_Typ);
  Hds[2] := Hd AND $0F;
  Hds[1] := Hd SHR 4;
  If HDs[1] = $F then HDs[1] := Read_Rtc(Rtc_HD1_extended);
  If HDs[2] = $F then HDs[2] := Read_Rtc(Rtc_HD2_extended);
  writeln('HD 1 : Typ ',Hds[1]);
  writeln('HD 2 : Typ ',Hds[2]);
end;

procedure Write_Memory;
{
 visualiza la memoria disponible
}
```

```
var base,extended : word;
begin;
  Base     := 256 * Read_Rtc(Rtc_Hi_Basememory) +
              Read_Rtc(Rtc_Lo_Basememory);
  extended := 256 * Read_Rtc(Rtc_Hi_Extendedmem) +
              Read_Rtc(Rtc_Lo_Extendedmem);
  writeln('Base memory: ',Base,' KB');
  writeln('Exteded memory: ',extended,' KB');
end;

procedure Write_Display;
{
 visualiza el tipo de la tarjeta gráfica, e informa si hay un
 coprocesador instalado
}
var dtyp : byte;
    Copro : byte;
begin;
  dtyp  := Read_Rtc(Rtc_Equipamiento);
  Copro := (dtyp AND 3) SHR 1;
  dtyp  := (dtyp AND 63) SHR 4;
  case dtyp of
    0 : begin;
          writeln('Extended functionality GFX-Controller');
        end;
    1 : begin;
          writeln('Color Display en modo 40 columnas');
        end;
    2 : begin;
          writeln('Color Display en modo 80 columnas');
        end;
    3 : begin;
          writeln('Monochrome Display Controller');
        end;
  end;
  if Copro = 1 then
    writeln('Coprocesador encontrado')
  else
    writeln('No se encontró coprocesador');
end;

procedure write_shadow;
{
 indica qué zonas son soportadas por la Shadow-Ram
}
var shadow : byte;
begin;
  shadow := read_rtc(Rtc_Opciones1);
  shadow := shadow AND 3;
```

```
  case shadow of
    0 : begin;
          writeln('Shadow System AND Video BIOS');
        end;
    1 : begin;
          writeln('Shadow System BIOS');
        end;
    2 : begin;
          writeln('Shadow disabled');
        end;
  end;
end;

procedure write_cpuspeed;
{
 indica si la CPU se encuentra en modo Turbo
}
var speed : byte;
begin;
  speed := read_rtc(Rtc_CPU_speed);
  if speed = 1 then
    writeln('CPU en modo Turbo')
  else
    writeln('CPU en modo Deturbo');
end;

var speed : byte;
begin;
 clrscr;
 Write_Floppy;
 Write_Hd;
 Write_Memory;
 Write_Display;
 Write_Shadow;
 Write_CPUSpeed;
end.
```

13. Descubra un mundo de sonidos - la tarjeta Sound-Blaster

La tarjeta Sound Blaster es actualmente el estándar de sonido en el PC. La tarjeta comercializada en 1989 por Creative Labs ha sufrido muchas modificaciones desde su versión primigenia. Después de la *SB Pro* y la *SB 16*, Creative ya ha llegado a la *AWE 32*. En principio todas las tarjetas son compatibles hacia abajo, lo que es cierto para el chip DSP, el procesador digital de señales, aunque con alguna limitación.

Sin embargo, en el chip OPL y el mezclador, se termina la compatibilidad. Por ello ha de incluir en sus programas un tratamiento diferencial de las tarjetas de sonido. Pero no tema, en el siguiente capítulo aprenderá todo lo que necesita para trabajar con Sound Blaster. Puede encontrar una detallada referencia de hardware así como su aplicación en la práctica.

13.1 Componentes de las tarjetas Sound-Blaster

Vamos a ocuparnos primero del hardware de la tarjeta Sound Blaster. Sin su conocimiento no es posible una programación efectiva de la tarjeta. El corazón de la Sound Blaster es el DSP, el procesador digital de señales. Tiene las funciones de un conversor *AD/DA* y es el responsable de la entrada y salida de los datos digitales.

Como segundo chip se puede encontrar en la tarjeta un chip OPL. Es el responsable de la síntesis *FM*. Ya que esta técnica hoy en día está algo anticuada, no vamos a extendernos sobre ella. A partir de la Sound Blaster Procedimiento también se puede encontrar en la tarjeta un chip mezclador. Este chip tiene funciones tan interesantes como la regulación del volumen o del balance.

Programar el procesador digital de señales - DSP

El DSP empleado en la serie de las Sound Blaster tiene la denominación *CT-DSP-1321*. Se encuentra en la dirección de port *2x0h*, donde la *x* representa la dirección base ajustada en cada momento. Se programa escribiendo un valor en uno de sus registros. La dirección base se puede elegir libremente en pasos de *10h* y puede estar entre *210h* y *280h*. Se dispone de los siguientes registros.

Port	Nombre	Estado
B +00h	Registro FM izq. selección y estado	R/W
B +01h	Registro FM izq. datos	W
B +02h	Registro FM dcho. selección y estado	R/W
B +03h	Registro FM dcho. datos	W
B +04h	Chip mezclador, registro de índice	W
B +05h	Chip mezclador, registro de datos	L/W
B +06h	Reset DSP	W
B +08h	Ambos registros FM, selección y estado	R/W
B +09h	Ambos registros FM, datos	W
B +0Ah	Datos DSP (Read Data)	R
B +0Ch	Datos DSP o comando y estado	R/W
B +0Dh	DSP Timer_Interrupt Clear Port	R
B +0Eh	Estado de datos DSP existentes	R
B +0Fh	DSP 16-bit Voice-Int. Clear Port	R
B +10h	Datos CD-ROM	R
B +11h	Comando CD-ROM	W
B +12h	Reset CD-ROM	W
B +13h	Activar CD-ROM	W
338h	FM Register Status-Port	R
339h	Registro de selección FM	W
38Bh	Selección registro de FM avanzado	W
38Bh	Registro de FM avanzado port de datos	W

Fundamentalmente, un acceso a los registros se realiza de la siguiente forma: primero se escribe en la Sound Blaster el número del comando y a continuación los parámetros necesarios mediante el port de comandos *2xCh*. Un acceso de lectura se realiza mediante el port de datos *2xAh*.

Pero antes de enviar alegremente comandos a la Sound Blaster, debería realizar primero un reset. Esto se puede conseguir mediante el port de reset *2x6h*. Para ello escriba el valor *1* en el port de reset, espere brevemente, y escriba ahora el valor *0* en el mismo port. Puede averiguar si el reset tuvo éxito, si al cabo de unos *100ms* el port de datos presenta el valor *$AA*. Si el valor se pudo encontrar, el reset funcionó, sino habrá que repetirlo. Puede encontrar un ejemplo de qué aspecto podría tener una función de reset en *Reset_SBCard*.

```
FUNCTION Reset_SBCard : BOOLEAN;
{
 La función reinicia el DSP. Si el reset tuvo éxito, se devuelve
 TRUE, sino FALSE
}
CONST  ready = $AA;
VAR ct, stat : BYTE;
BEGIN
  PORT[dsp_adr + $6] := 1;          { dsp_adr + $6 = función de reset}
```

```
  FOR ct := 1 TO 100 DO;
  PORT[dsp_adr + $6] := 0;
  stat := 0;
  ct   := 0;                          { La comparación ct < 100, ya }
  WHILE (stat <> ready)               { que la inicialización dura  }
  AND   (ct < 100)       DO BEGIN     { aprox. 100 ms.              }
    stat := PORT[dsp_adr + $E];
    stat := PORT[dsp_adr + $a];
    INC(ct);
  END;
  Reset_SBCard := (stat = ready);
END;
```

El hecho de que en el caso de un reset con éxito se pueda encontrar el valor *0AAh* en el registro de datos, se puede aprovechar para reconocer el port base. En un bucle simplemente se comprueban todos los ports posibles desde *210h* hasta *280h* en pasos de *10h* como candidatos potenciales de port base de la Sound Blaster. La función *Detect_SBReg* representa un ejemplo de cómo podría realizarse una rutina de detección. La función emplea lo expuesto y devuelve *TRUE* si se pudo encontrar una tarjeta Sound Blaster.

```
FUNCTION Detect_SBReg : BOOLEAN;
{
 La función devuelve TRUE si se pudo inicializar una SoundBlaster, de
 lo contrario FALSE. La variable dsp_adr se fija en la dirección
 base de la SB.
}
VAR
  Port, Lst : WORD;
BEGIN
  Detect_SBReg := SbRegDetected;
  IF SbRegDetected THEN EXIT;          { Exit, si inicializado       }
  Port := Startport;                   { Posibles direcc. SB entre   }
  Lst  := Endport;                     { $210 y $280 !               }
  WHILE (NOT SbRegDetected)
  AND   (Port <= Lst)  DO BEGIN
    dsp_adr := Port;
    SbRegDetected := Reset_SBCard;
    IF NOT SbRegDetected THEN
      INC(Port, $10);
  END;
  Detect_SBReg := SbRegDetected;
END;
```

Ahora ya ha encontrado su tarjeta Sound Blaster y sabe en qué port se puede encontrar. Pero con ello hemos llegado al siguiente problema: ¿Cuál de las múlti-

ples versiones de Sound Blaster estará presente? Al fin y al cabo, el control de una SB16 es sensiblemente diferente que el de una SB normal.

Para averiguar esto, emplearemos una función que nos ofrece la Sound Blaster: la obtención del número de versión de la SB. Para ello hemos de enviar el comando a la SB, y como contrapartida obtenemos el número de versión en el registro de datos. ¿Pero cómo se escribe un comando en la SB?

En principio basta con saber que los comandos siempre se envían mediante el port de comandos *2xCh*. Sin embargo se ha de tener en cuenta de que no se puede escribir en el port mientras el bit 7 del port de comandos está activado. Así que ha de esperar en un bucle hasta que el port esté listo para escritura y después podrá enviar su valor al port. Aquí tiene un ejemplo de cómo se podría realizar el procedimiento.

```
procedure Wr_dsp(v : byte);
{
 Espere hasta que el DSP esté listo para escribir el byte
 pasado en "v" en él.
}
begin;
  while port[dsp_adr+$c] >= 128 do ;
  port[dsp_adr+$c] := v;
end;
```

Pero como no sólo queremos escribir valores a la Sound Blaster, sino también leer valores de ella, necesitamos una función que nos lea un valor de la SB. Para leer un valor del port de datos, se ha de esperar hasta que éste ya no tenga el valor *0AAh*. Entonces se puede efectuar la lectura. Veamos este simple proceso en la práctica: La función *SBReadSB* trabaja según el procedimiento anteriormente descrito y le devuelve el byte actual del port de datos.

```
FUNCTION SbReadByte : BYTE;
{
 La función espera hasta que el DSP se pueda leer, y devuelve
 el valor obtenido
}
begin;
  while port[dsp_adr + $a] = $AA do ;      { Esperar, hasta que DSP
listo}
  SbReadByte := port[dsp_adr + $a];        { Escribir valor
}
end;
```

Pero volvamos a nuestro problema. Queremos determinar el número de versión de la Sound Blaster. Para ello empleamos el comando *$E1*, que sirve para la obtención de la versión. Después de enviar este comando se puede leer el número de

versión en el port de datos. El número se compone de dos bytes. Primero se devuelve el número de versión principal. Mediante una nueva lectura del port se obtiene el número de versión secundario. *SBGetDSPVersion* es un procedimiento que realiza todas estas funciones.

```
PROCEDURE SbGetDSPVersion;
{
Obtiene la versión del DSP y guarda el resultado en las variables
globales
 SBVERSMAJ y SBVERSMIN así como SBVERSSTR.
}
VAR i : WORD;
    t : WORD;
    s : STRING[2];
BEGIN
  Wr_dsp($E1);                          { $E1 = Consulta de versión  }
  SbVersMaj := SbReadByte;
  SbVersMin := SbReadByte;
  str(SbVersMaj, SbVersStr);
  SbVersStr := SbVersStr + '.';
  str(SbVersMin, s);
  if SbVersMin > 9 then
    SbVersStr := SbVersStr +          s
  else
    SbVersStr := SbVersStr + '0' + s;
END;
```

Además del comando para el reconocimiento de la versión la Sound Blaster naturalmente posee toda una serie de otros comandos. Aquí presentamos un resumen de los comandos del DSP de la tarjeta SB.

Comando	Función
10h	Salida directa de 8 bits
14h	Salida de 8 bits mediante DMA
16h	Salida de muestreos de 2 bits comprimidos mediante DMA
17h	Salida de muestreos de 2 bits comprimidos mediante DMA con byte de referencia
20h	Grabación directa
24h	Grabación de muestreos de 8 bits mediante DMA
30h	Entrada MIDI directa
31h	Entrada MIDI mediante interrupción
32h	Entrada MIDI directa con Timestamp
33h	Entrada MIDI mediante interrupción con Timestamp
34h	Modo MIDI-UART, directo
35h	Modo MIDI-UART, mediante interrupción
37h	MIDI-UART mediante interrupción con Timestamp
38h	Enviar código MIDI
40h	Fijar tasa de muestreos
45h	DMA-Multiblock Continue SB16
48h	Ajustar tamaño de bloque
...	

Comando	Función
...	
74h	Salida de muestreos de 4 bits comprimidos vía DMA
75h	Salida de muestreos de 4 bits comprimidos con byte de referencia vía DMA
76h	Salida de muestreos de 2,6 bits comprimidos vía DMA
77h	Salida de muestreos de 2,6 bits comprimidos con byte de referencia vía DMA
80h	Definir bloque de silencio
91h	Salida de 8 bits mediante DMA en High-Speed
99h	Entrada de 8 bits mediante DMA en High-Speed
B6h	Reproducir datos de 16 bits en la SB16 vía DMA
B9h	Grabar datos de 16 bits en la SB16 vía DMA
C6h	Reproducir datos de 8 bits en SB16 vía DMA
C9h	Grabar datos de 8 bits en la SB16 vía DMA
D0h	Detener DMA
D1h	Activar altavoz
D3h	Apagar altavoz
D4h	Continuar DMA
D8h	Obtener ajuste de altavoz
E1h	Consulta de versión

Veamos un poco más de cerca los diferentes comandos:

10h - Salida directa de 8 bits

Para reproducir datos periódicamente mediante la interrupción del temporizador, emplee la función *10h*. ¡Ambos pasos se han de repetir con velocidad constante!

- Envíe el comando *10h*.
- Envíe el byte de datos.

14h - Salida de 8 bits mediante DMA

La salida mediante DMA es bastante compleja - al menos para alguien que no se haya ocupado anteriormente de esta temática. Es muy importante que esté en disposición de programar directamente el controlador DMA, ya que sino obtendrá muchos efectos, pero no la reproducción de los datos de muestreos. Puede encontrar más informaciones sobre el controlador DMA y su programación en el apartado 12.4.

Así que si quiere reproducir datos mediante DMA, ha de desviar primero la interrupción Sound Blaster a una rutina propia y programar correctamente el controlador DMA. A continuación ha de determinar la tasa de muestreo. Después se ha de proceder de la siguiente forma:

- Envíe el comando *14h*.
- Envíe el byte bajo de la longitud del muestreo -1.

- Envíe el byte alto de la longitud de muestreo -1.

La reproducción se inicia en cuanto haya realizado el tercer paso. Tenga en cuenta que no puede reproducir bloques mayores de 64 Kbytes. Los bloques de este tamaño se han de dividir en varios subbloques.

En la práctica debería proceder de la siguiente forma: para comenzar calcule la dirección física del bloque a reproducir. Esto se hace con la fórmula

```
Physpos := 16 * sgm(Bloque^)+ofs(Bloque^).
```

La palabra alta de la dirección física es la página que le ha de pasar al controlador DMA. La palabra baja contiene el offset que ha de fijar. Bloquee primero el canal DMA, para evitar accesos. Después borre el Flip-Flop y envíe el modo de escritura del controlador DMA. Para la salida de datos necesita el modo *$48 + número* del canal DMA, es decir, *$49* si emplea el canal DMA *1*. A continuación fije primero el byte bajo y después el byte alto del offset de la dirección física. Después envíe el byte bajo y a continuación el byte alto del tamaño del bloque a transferir, antes enviar la página de la dirección física. Con ello el controlador DMA está listo para la transferencia y podemos dedicarnos a la programación de la tarjeta Sound Blaster. Envíe primero el comando *$14* para la salida de 8 bits mediante DMA. Envíe después el byte bajo y el byte alto del tamaño del bloque a transferir. Con ello la Sound Blaster está programada. Libere ahora el canal DMA y la reproducción de sonido comienza.

Puede encontrar un ejemplo de cómo funciona la programación correctamente en el procedimiento *Reprod_SB*. Le pasamos como primer parámetro la página de la dirección física del bloque de muestreos. El segundo parámetro contiene la parte de offset de la dirección física del bloque de muestreos y el tercero especifica su longitud.

```
procedure Reprod_Sb(Segm, Offs, dsize : word);
{
 Este procedimiento reproduce el bloque direccionado mediante
segm:offs con el tamaño dsize. Se ha de tener en cuenta que el con-
trolador DMA NO puede trabajar saltando páginas...
}
var li : word;
begin;
  port[$0A] := dma_ch + 4;                { Bloquear canal DMA        }
  Port[$0c] := 0;                         { Direcc. del buffer  (blk) }
  Port[$0B] := $48 + dma_ch;              { para salida de sonido     }
  Port[dma_adr[dma_ch]] := Lo(offs);      { al controlador DMA        }
  Port[dma_adr[dma_ch]] := Hi(offs);
  Port[dma_wc[dma_ch]] := Lo(dsize-1); { Tamaño del bloque al         }
  Port[dma_wc[dma_ch]] := Hi(dsize-1); { controlador DMA              }
  Port[dma_page[dma_ch]] := Segm;
```

```
    Wr_dsp($14);
    Wr_dsp(Lo(dsize-1));                { Tamaño del bloque al      }
    Wr_dsp(Hi(dsize-1));                { DSP                       }
    Port[$0A] := dma_ch;                { Liberar canal DMA         }
end;
```

16h - Salida de muestreos de 2 bits comprimidos, vía DMA

La reproducción de muestreos de 2 bits comprimidos se realiza de forma análoga a la reproducción de muestreos de 8 bits. Simplemente ha de sustituir el comando *$14* por el comando *$16* para la programación de la Sound Blaster.

17h - Salida de muestreos de 2 bits comprimidos con byte de referencia vía DMA

Para el comando *17h* se emplea lo dicho para el comando *16h*. La única diferencia es que el primer byte se interpreta como byte de referencia. Es casi un valor inicial, que se forma de las diferencias. Habitualmente se ha de enviar primero un bloque con byte de referencia mediante el comando *17h*, al que siguen los bloques sin byte de referencia.

20h - Grabación directa

El comando *20h* sirve para la grabación directa de datos de muestreos de 8 bits. Si quiere grabar mediante este comando, el modo de proceder es el siguiente:

- Enviar comando *20h*.
- Leer byte de muestreos.

¡Tenga en cuenta que los dos pasos se han de repetir periódicamente, con velocidad constante! Este método sólo es adecuado para la grabación con tasas de muestreos bajas. Para grabaciones de mayor calidad se ha de emplear el método DMA.

24h - Grabación de muestreos de 8 bits mediante DMA

Si quiere grabar datos mediante DMA, ha de desviar primero la interrupción de la Sound Blaster a una rutina propia, y programar el controlador DMA adecuadamente a continuación. Después se ha de determinar y fijar la tasa de muestreos. Ahora ha de proceder de la siguiente forma:

- Envíe el comando *24h*.
- Envíe el byte bajo de la longitud de muestreos -1.
- Envíe el byte alto de la longitud de muestreos -1.

La reproducción se inicia en cuanto haya finalizado el tercer paso. Tenga en cuenta que no puede aceptar bloques mayores de 64 Kbytes. Los bloques de este tamaño se han de dividir en varios subbloques. En la práctica debería proceder de la siguiente forma: primero ha de calcular la dirección física del bloque a reproducir. Esto se realiza según la fórmula

```
Physpos := 16 * sgm(Bloque^)+ofs(Bloque^).
```

La palabra alta de la dirección física es la página que le ha de pasar al controlador DMA. La palabra baja contiene el offset que ha de fijar. Bloquee primero el canal DMA, para evitar accesos. Después borre el Flip-Flop y envíe el modo de escritura del controlador DMA. Para la salida de datos necesita el modo *$44 + número* del canal DMA, es decir, *$45* si emplea el canal DMA *1*. A continuación fije primero el byte bajo y después el byte alto del offset de la dirección física. Después envíe el byte bajo y a continuación el byte alto del tamaño del bloque a transferir, antes enviar la página de la dirección física. Con ello el controlador DMA está listo para la transferencia y podemos dedicarnos a la programación de la tarjeta Sound Blaster. Envíe primero el comando *$24* para la salida de 8 bits mediante DMA. Envíe después el byte bajo y el byte alto del tamaño del bloque a transferir. Con ello la Sound Blaster está programada. Libere ahora el canal DMA y la grabación de sonido comienza.

Puede encontrar un ejemplo de cómo funciona la programación correctamente en el procedimiento *Graba_SB*. Le pasamos como primer parámetro la página de la dirección física del bloque de muestreos. El segundo parámetro contiene la parte de offset de la dirección física del bloque de muestreos y el tercero especifica su longitud.

```
procedure Graba_Sb(Segm, Offs, dsize : word);
{
 Este procedimiento graba en el bloque direccionado con segm:offs con
el tamaño dsize. Se ha de tener en cuenta que el controlador DMA NO
puede trabajar saltando páginas...
}
var li : word;
begin;
  port[$0A] := dma_ch + 4;                  { Bloquear canal DMA          }
  Port[$0c] := 0;                           { Direcc. del buffer  (blk)   }
  Port[$0B] := $44 + dma_ch;                { para salida de sonido       }
  Port[dma_adr[dma_ch]] := Lo(offs);        { al controlador DMA          }
  Port[dma_adr[dma_ch]] := Hi(offs);
  Port[dma_wc[dma_ch]] := Lo(dsize-1);      { Tamaño del bloque al        }
  Port[dma_wc[dma_ch]] := Hi(dsize-1);      { controlador DMA             }
  Port[dma_page[dma_ch]] := Segm;
```

```
    Wr_dsp($14);
    Wr_dsp(Lo(dsize-1));                    { Tamaño del bloque al      }
    Wr_dsp(Hi(dsize-1));                    { DSP                       }
    Port[$0A] := dma_ch;                    { Liberar canal DMA         }
  end;
```

30h - Entrada MIDI directa

El comando *30h* sirve para la grabación directa de datos MIDI. El modo de proceder es el siguiente:

- Escribir *30h* al DSP.
- Leer port de estado hasta que el bit 7 esté activo.
- Leer datos del port de datos.

Si el bit 7 del byte de datos está activo, se trata de un byte de control. Los pasos 2 y 3 se han de repetir para llegar a la nota. Si no, esta está directamente disponible.

31h - Entrada MIDI mediante interrupción

Para que no tenga que estar comprobando constantemente en su programa si hay presentes datos en el port MIDI, la Sound Blaster dispone del comando *31h*. Con él puede leer los datos MIDI controlado por interrupciones. Para ello ha de desviar primero la interrupción Sound Blaster a una rutina propia. A continuación envíe el comando *31h*. Si ahora hay datos MIDI presentes, se dispara la interrupción Sound Blaster y puede evaluar los datos. Para ello proceda como sigue:

- Leer byte de datos del port de datos y guardarlo.
- Leer port de estado.
- Finalizar interrupción.

32h - Introducción MIDI directa con Timestamp

Para poder temporizar los datos MIDI con exactitud, existe la posibilidad del Timestamp. Un Timestamp es un valor de 24 bits, que contiene el número de milisegundos que han transcurrido desde el envío del comando *32h*. Proceda de la siguiente forma:

- Escriba el comando *32h*.
- Compruebe si el bit 7 del port de estado está activo (tiene el valor 1).
- Lea el byte bajo del Timestamp.
- Lea el byte medio del Timestamp.
- Lea el byte alto del Timestamp.
- Lea el código MIDI.

Los pasos 2 hasta 6 se repiten continuamente.

33h - Entrada MIDI I con Timestamp mediante la interrupción

El comando trabaja igual que el comando *31h*. Sólo que aquí ha de leer primero los tres bytes del Timestamp, antes de poder leer el código MIDI.

34h - Modo MIDI-UART, directo

Después de haber activado este modo, puede enviar y recibir datos MIDI directamente. Ya no ha de enviar el comando *38h*, sino que puede escribir directamente en el DSP.

35h - Modo MIDI-UART mediante interrupción

Este comando provoca en principio lo mismo que el comando *34h*, pero está controlado por interrupciones. Puede encontrar un ejemplo de su procesamiento bajo el comando *32h*.

37h - MIDI-UART mediante interrupción con Timestamp

También aquí el proceso de los datos se realiza mediante la interrupción. La estructura de la rutina corresponde a la del comando *35h*, sólo que aquí se han de leer primero los tres bytes del Timestamp, antes de poder obtener el código MIDI.

38h - Enviar código MIDI

Para enviar un código MIDI para el control de un dispositivo conectado, proceda de la siguiente forma:

- Envíe el comando *38h*.
- Lea el port de comando hasta que el bit 7 no está a *1*.
- Escriba su código MIDI.

40h - Fijar tasa de muestreos

Este comando es muy importante. Con él determina la tasa de muestreos de la tarjeta Sound Blaster. Así que necesitará este comando en todos los programas en los que emplee salida digital. Proceda de la siguiente forma:

- Envíe el comando *40h*.
- Envíe la constante TC calculada.

Esta constante TC se calcula de forma diferente para los modos de velocidad normal (Normal Speed) y los de alta velocidad (High Speed, a partir de Sound Blaster Procedimiento). Para los modos de velocidad normal puede emplear la siguiente fórmula:

```
TC := 256 - (1.000.000 DIV frecuencia)
```

Para el empleo del modo High-Speed:

```
TC := 65536 - (256.000.000 DIV frecuencia)
```

Importante: ¡sólo se envía el byte alto de la constante TC calculada, el byte bajo se ignora!

45h - DMA Multi-Block Continue SB16

En una SB16 tiene la posibilidad de continuar la transferencia de un bloque con este comando, en vez de los comandos *B6h, B9h, C6h, C9h.* Así no ha de volver a indicar la longitud y el modo de transferencia, sino que puede enviar directamente el comando *45h.* Se envía un bloque con el mismo tamaño que el transferido anteriormente.

48h - Ajustar tamaño de bloque

Mediante este comando ajusta el tamaño del bloque de muestreos en el modo High-Speed. Para ello ha de

- enviar primero el comando *48h,*
- enviar el byte bajo del tamaño del bloque -1,
- enviar el byte alto del tamaño del bloque -1.

74h - Salida de muestreos comprimidos de 4 bits vía DMA

Para la salida de muestreos de 4 bits comprimidos se emplea el mismo método que se describió bajo el comando *14h.*

75h - Salida de muestreos comprimidos de 4 bits con byte de referencia vía DMA

Vea para ello las explicaciones del comando *17h.*

76h - Salida de muestreos comprimidos de 2,6 bits vía DMA

Véase para ello el comando *74h.*

77h - Salida de muestreos comprimidos de 2,6 bits con byte de referencia vía DMA

Véase para ello el comando *75h.*

80h - Definir bloque de silencio

Mediante este comando puede definir y reproducir un bloque de silencio. Proceda de la manera descrita:

- Fije la interrupción de la Sound Blaster en su propia rutina.

- Fije la tasa de muestreos (comando *40h*).
- Envíe el comando *80h*.
- Envíe el byte bajo del tamaño de bloque -1.
- Envíe el byte alto del tamaño de bloque -1.

Cuando se haya reproducido completamente el bloque de silencio, la Sound Blaster genera una interrupción.

91h - Salida de 8 bits mediante DMA en el modo de alta velocidad

Mediante este comando se pueden reproducir datos de muestreos con una calidad de hasta 44 kHz. Para ello ha de proceder de forma similar que en la reproducción normal:

- Desvíe la interrupción SB a su propia rutina.
- Inicialice los controladores DMA.
- Fije la tasa de muestreos mediante el comando *40h*.
- Envíe *48h* para ajustar el tamaño del bloque.
- Envíe el byte bajo del tamaño del bloque -1.
- Envíe el byte alto del tamaño del bloque -1.
- Envíe el comando *91h*.

La reproducción de los datos se inicia inmediatamente después de haber escrito el comando *91h*. Al final de la transferencia la tarjeta Sound Blaster genera una interrupción.

99h - Entrada de 8 bits mediante DMA en el modo de alta velocidad

La grabación de datos en el modo High-Speed se realiza igual que la reproducción de los mismos. La única diferencia es que al final no ha de enviar el comando *91h*, sino *99h*.

B6h - reproducir datos de 16 bits vía DMA

A partir de la SB16 también tiene la posibilidad de samplear y reproducir datos de 16 bits. La sistemática es la misma que en la reproducción convencional de datos mediante DMA, aunque en la Sound Blaster 16 ya no se ha de ajustar mediante el chip mezclador si se han de transferir datos en mono o estéreo. Esto ahora se puede comunicar durante la inicialización del DSP. Proceda de la siguiente manera.

- Desvíe la interrupción de la SB a su propia rutina.
- Inicialice el controlador DMA.
- Envíe el comando *B6h*.

- Envíe *00h* para la reproducción de datos mono, o *20h* para datos en estéreo.
- Envíe el byte bajo del tamaño del bloque -1.
- Envíe el byte alto del tamaño del bloque -1.

La reproducción se inicia inmediatamente después de enviar la longitud de bloque. Si ha finalizado la reproducción de los datos, la Sound Blaster (teóricamente) genera una interrupción.

B9h - Grabar datos de 16 bits en la SB16 vía DMA

La reproducción de datos de 16 bits se realiza de la misma forma que la reproducción de los mismos. La única diferencia es que ha de enviar el comando *B9h* en vez de *B6h*.

C6h - Reproducir datos de 8 bits en SB16 vía DMA

La reproducción de datos de 8 bits en la SB16 se realiza de la misma forma que la de datos de 16 bits. La diferencia estriba en que ha de emplear el comando *C6h* y no *B6h*. Véase también para ello el comando *B6h*.

C9h - Grabar datos de 8 bits en la SB16 vía DMA

Aquí se aplica lo mismo que en el comando *B9h*, sólo que ha de emplear el comando *C9h*.

D0h - Detener DMA

Mediante este comando puede interrumpir la transferencia de datos vía DMA. Mediante el comando *D4h* se puede continuar una reproducción interrumpida.

D1h - Activar altavoz

Para poder escuchar los datos reproducidos, primero ha de activar el altavoz de la tarjeta Sound Blaster. Esto se puede obtener, enviando el comando *D1h*.

D3h - Apagar altavoz

Mediante este comando vuelve a apagar el altavoz de la tarjeta Sound Blaster.

D4h - Continuar DMA

Mediante este comando puede continuar una reproducción DMA interrumpida por el comando *D0h*.

D8h - Consultar ajuste de altavoz

Para averiguar si el altavoz de la SB ya está activado, a partir de la SB Pro dispone de este comando. Mediante él obtiene el estado del altavoz. Para ello escriba el comando *D8h* y lea a continuación un byte del DSP. Si tiene el valor *$FF* el altavoz está activado. Si por lo contrario es *1*, está apagado.

E1h - Consulta de versión

Puede averiguar el número de versión de la tarjeta Sound Blaster mediante este comando.

- Envíe el comando *E1h*,
- lea el número versión principal de la SB,
- lea el número secundario de la SB.

Reconocimiento de la tarjeta Sound-Blaster

Cuando instala la tarjeta Sound Blaster, el programa de instalación incluye la variable de entorno BLASTER en el AUTOEXEC.BAT. Con ayuda de esta variable puede reconocer los parámetros de la SB en la mayoría de los casos de forma simple y sin riesgos. ¿Pero qué hacemos si el usuario quiere escuchar sonido, pero la variable de entorno no está fijada? Muy sencillo, intentamos reconocer la tarjeta por hardware. Este método sin embargo tiene sus limitaciones. Es bastante "peligroso" realizar un test de tarjeta, ya que a las tarjetas de red o chips S3 posiblemente instalados no le hace nada de gracia que alguien escriba "salvajemente" por sus registros. Atención, las tarjetas no se dañan, pero sí puede ocurrir que el ordenador se quede colgado. El port base de la SB se puede reconocer con bastante sencillez, pero en las interrupciones ha emplear algunos trucos. Ante el canal DMA sólo nos queda capitular, y pedirle al usuario que lo introduzca él, si no se encuentra en *1* (lo que suele ser en la mayoría de los casos).

¿Cómo se reconoce el port base de la tarjeta Sound Blaster? Muy sencillo, las direcciones posibles de la SB se encuentran entre *200h* y *280h* y se pueden variar en pasos de *10h*. Así que sólo hemos de comprobar todas las direcciones para ver si en ellas se encuentra una SB. Si es el caso, todo queda claro. Si no se encontró ninguna Sound Blaster hasta la dirección *280h*, este nos debería dar que pensar. O bien no hay ninguna Sound Blaster instalada en el ordenador (a lo mejor se trata de una Gravis Ultrasound con SBOS), o bien nuestras rutinas de reconocimiento han fallado. En el segundo caso podemos excluir en principio que se indicó el canal DMA correcto. Si este es erróneo, el reconocimiento sólo puede fallar. ¿Así que, cómo saber que en la dirección actual se encuentra una Sound Blaster? Para averiguar esto, hemos de intentar realizar un reset. Si este funciona, en la dirección se encuentra una Sound Blaster, de lo contrario hemos de seguir buscando.

Las rutinas necesitadas para este reconocimiento, *Reset_SBCard* y *Detect_SBReg*, ya las ha conocido. Ahora que ya conocemos el port base de la tarjeta Sound Blaster, podemos averiguar la interrupción. El planteamiento es el siguiente: cuando transferimos un bloque mediante DMA, siempre se dispara una interrupción al final de la transmisión. Así que nuestra rutina de test de interrupciones enviará un pequeño bloque por DMA, esperando a continuación para darle tiempo a la Sound Blaster de dispara una interrupción. Esto lo repetimos en un bucle. En este bucle siempre desviamos el candidato potencial de interrupción a nuestra propia rutina. Cuando se salta a ésta, allí activamos una bandera conforme hemos encontrado la interrupción. En el bucle podemos comprobar si esta bandera está activada. Si es así, hemos encontrado la interrupción y podemos salir del bucle. Por el contrario hemos de seguir.

La rutina *Detect_SbIRQ* demuestra la aplicación de la teoría. Recuerde guardar los antiguos valores de interrupción, para poder restaurarlos después de procesar esta rutina. También debería apagar el altavoz antes de la transferencia del bloque, ya que no es muy elegante dejar que el usuario escuche la basura de datos que enviamos para realizar el test.

```
procedure detect_sbIRQ;
{
 Esta rutina reconoce la IRQ de la tarjeta SoundBlaster. Se compru-
 eban todas las posibles interrupciones. Para ello se envian bloques
 cortos via DMA. Si al final de la salida se salta a la interrupción
 ajustada, es que se encontró la correcta.
}
const IRQs_Posibles : array[1..5] of byte = ($2,$3,$5,$7,$10);
var i : integer;
    h : byte;
begin;
    getintvec($8+dsp_irq,intback);          { guardar valores!          }
    port21 := port[$21];
    fillchar(buffer1^,1200,128);
    set_Timeconst_sb16(211);
    wr_dsp_sb16($D3);                       { apagar altavoz            }
    i := 1;
    interrupt_check := true;
    while (i <= 5) and (not IRQDetected) do
    begin;
    dsp_irq := IRQs_Posibles[i];            { IRQ a probar              }
    getintvec($8+dsp_irq,oldint);           { desviar interrupción      }
    setintvec($8+dsp_irq,@Dsp_Int_sb16);
    irqmsk := 1 shl dsp_irq;
    port[$21] := port[$21] and not irqmsk;
    Sampling_Rate := 211;
```

```
Tam_Bloque := 1200;                    { salida a prueba          }
dsp_block_sb16(Tam_Bloque,Tam_Bloque,bufferl,true,false);
delay(150);
setintvec($8+dsp_irq,oldint);          { Interrupción de vuelta   }
port[$21] := Port[$21] or irqmsk;
h := port[dsp_adr+$E];
Port[$20] := $20;
inc(i);
end;
interrupt_check := false;
wr_dsp_sb16($D1);                      { encender altavoz         }
setintvec($8+dsp_irq,intback);         { antiguos valores de vuelta}
port[$21] := port21;
dsp_rdy_sb16 := true;
end;
```

Mezclar datos de sonido - chip mezclador

La tarjeta Sound Blaster posee, desde la versión SB Pro un chip mezclador aparte del DSP. Este suele ser responsable de la regulación de las entradas y salidas de la tarjeta Sound Blaster. Además también controla el volumen y el balance. El chip mezclador de la SB 16 es más o menos compatible hacia abajo (el hecho de la compatibilidad aparentemente se volvió a tratar de aquella manera), pero se ha de programar a través de otros registros, si se quieren emplear sus funciones ampliadas.

Se direcciona mediante la pareja de registros *2x4h* y *2x5h*. *2x4h* es el port de selección del mezclador y *2x5h* el port de datos.

Si quiere programar el mezclador, ha de elegir el registro a modificar mediante el port de selección para poder leer o escribir a continuación datos en el port.

Echemos un vistazo al chip mezclador de la SB Pro:

Registro 00h - Reset

Este registro sirva para colocar el chip mezclador en su estado por defecto. Para ello proceda de la siguiente forma:

- Escriba el valor *0* al registro de índice.
- Espere aprox. 100 ms.
- Escriba el valor *0* al registro de datos.

Registro 02h - Volumen DSP

Este registro sirve para ajustar el volumen del DSP. Los bits 7 hasta 5 representan el volumen del canal izquierdo. De modo que el volumen puede estar entre *0* y *7*. En consecuencia, los bits 3 hasta 1 representan al volumen del canal derecho. Para determinarlo, proceda de la siguiente manera:

- Escriba el valor *02h* al registro de índice.
- Calcule el volumen para la SB Pro según:

```
Volumen := (left shl 5) + (right shl 1);
```

- Escriba el volumen al registro de datos.

Registro 0Ah - Volumen de micrófono

Este registro sirve para ajustar el volumen del micrófono. Se permiten valores entre *0* y *3*. El volumen se fija así:

- Escriba el valor *0Ah* al registro de índice.
- Calcule el volumen según:

```
Volumen := (Valor shl 1);
```

- Escriba el volumen al registro de datos.

Registro 0Ch - Ajustes de filtro de entrada

Este registro regula los ajustes para la grabación. Se pueden elegir un filtro y la fuente de entrada. Si el bit 5 no está activo, el filtro está activo, de lo contrario está apagado. El bit 3 regula la banda de paso del filtro. Si el bit 3 está activo es alta, sino es baja. Los bits 2 y 1 regulan la selección de la fuente de entrada. *0* representa al micrófono, *1* la entrada desde CD-ROM y *3* a Line Input. Para fijar este registro se puede proceder de la siguiente forma:

- Escriba el valor *$0C* en el registro de índice.
- Calcule el valor a ajustar:

```
Si filtro activo, entonces Ew := (1 shl 5)
Si es pasabajos, entonces Ew := Ew + (1 shl 3)
Ew := Ew + (Fuente shl 1)
```

- Escriba el valor de ajuste en el registro de datos.

Registro 0Eh - Filtro de salida y selección de estéreo

Este registro tiene dos funciones. Por una parte sirve para determinar si se ha deseleccionar un filtro de salida. Por la otra, conmuta entre reproducción mono y estéreo.

Si el bit 5 no está activo, el filtro está activo, de lo contrario está apagado. Si el bit 1 está a *1*, la reproducción se realizà en estéreo, sino en mono. Puede ajustar el registro de la siguiente forma:

- Escriba el valor *$0Eh* en el registro de índice.

```
Si con filtro entonces valor Reg := (1 shl 5)
Si estéreo entonces valor Reg := valor Reg + (1 shl 1)
```

- Escriba el valor de registro en el registro de datos.

Registro 22h - Volumen maestro

Este registro sirve para ajustar el volumen general. Los bits 7 hasta 5 representan al volumen del canal izquierdo. Por consiguiente el volumen puede adoptar valores de *0* a *7* en la SB Procedimiento. De forma análoga, los bits 3 hasta 1 representan al volumen para el canal derecho. Para determinar el volumen puede proceder de la siguiente forma:

- Escriba el valor *22h* en el registro de índice.
- Calcule el volumen para la SB Pro:

```
Volumen = (left shl 5) + (right shl 1)
```

- Escriba el volumen al registro de datos.

Registro 26h - Volumen FM

Este registro sirve para ajustar el volumen del canal FM. De lo contrario se aplica lo dicho sobre el registro *22h*. Este volumen se puede ajustar de la siguiente forma:

- Escriba el valor *26h* en el registro de índice.
- Calcule el volumen según

```
Volumen = (left shl 5) + (right shl 1)
```

- Escriba el volumen al registro de datos.

Registro 28h - Volumen del CD

Este registro sirve para ajustar el volumen del canal CD. De lo contrario se aplica lo dicho para el registro *22h*. Para determinar el volumen proceda así:

- Escriba el valor *28h* al registro de índice.
- Calcule el volumen para la SB Pro:

```
Volumen = (left shl 5) + (right shl 1)
```

- Escriba el volumen del registro de datos.

Registro 2Eh - Volumen Line

Este registro sirve para ajustar el volumen del canal de línea. De lo contrario se aplica lo dicho para el registro *22h*. Proceda así:

- Escriba el valor *2Eh* al registro de índice.
- Calcule el volumen según la fórmula:

```
Volumen = (left shl 5) + (right shl 1)
```

- Escriba el volumen al registro de datos.

Ahora que ya conoce el chip de mezcla de la SB Pro, el chip mejorado de la Sound Blaster 16 no debería presentarle ningún problema. Aquí un resumen de sus registros:

Registro 48: Left Master Volume		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	Master Volume	0
bit 4:	Master Volume	0
bit 5:	Master Volume	0
bit 6:	Master Volume	1
bit 7:	Master Volume	1

Registro 49: Right Master Volume		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	Master Volume	0
bit 4:	Master Volume	0
bit 5:	Master Volume	0
bit 6:	Master Volume	1
bit 7:	Master Volume	1

Register 50: Left Voice Volume		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Volume	0
bit 5:	Volume	0
bit 6:	Volume	1
bit 7:	Volume	1

Register 51: Right Voice Volume		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Volume	0
bit 5:	Volume	0
bit 6:	Volume	1
bit 7:	Volume	1

Registro 52: Left MIDI Volume		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	MIDI Volume	0
bit 5:	MIDI Volume	0
bit 6:	MIDI Volume	1
bit 7:	MIDI Volume	1

Register 53: Right MIDI Volume		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	MIDI Volume	0
bit 5:	MIDI Volume	0
bit 6:	MIDI Volume	1
bit 7:	MIDI Volume	1

Register 54: Left CD Volume		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Volume	0
bit 5:	Volume	0
bit 6:	Volume	0
bit 7:	Volume	0

Registro 55: Right CD Volume		
Nº bit	Función	Valor por defecto
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Volume	0
bit 5:	Volume	0
bit 6:	Volume	0
bit 7:	Volume	0

Register 56: Left Line Volume		
Nº bit	Función	Valor por defecto
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Volume	0
bit 5:	Volume	0
bit 6:	Volume	0
bit 7:	Volume	0

Register 57: Right Line Volume		
Nº bit	Función	Valor por defecto
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Volume	0
bit 5:	Volume	0
bit 6:	Volume	0
bit 7:	Volume	0

Registro 58: Mic Volume		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Volume	0
bit 5:	Volume	0
bit 6:	Volume	0
bit 7:	Volume	0

Register 59: Pc Speaker Volume		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	---	0
bit 5:	---	0
bit 6:	Volume	0
bit 7:	Volume	0

Register 60: Output Controll Register		
Nº bit	**Función**	**Valor por defecto**
bit 0:	Mic	1
bit 1:	CD derecha	1
bit 2:	CD izquierda	1
bit 3:	Line derecha	1
bit 4:	Line izquierda	1
bit 5:	---	0
bit 6:	---	0
bit 7:	---	0

Registro 61: Left In Controll Register		
Nº bit	**Función**	**Valor por defecto**
bit 0:	Mic	1
bit 1:	CD derecha	0
bit 2:	CD izquierda	1
bit 3:	Line derecha	0
bit 4:	Line izquierda	1
bit 5:	MIDI derecha	0
bit 6:	MIDI izquierda	0
bit 7:	- - -	0

Register 62: Right In Controll Register		
Nº bit	**Función**	**Valor por defecto**
bit 0:	Mic	1
bit 1:	CD derecha	1
bit 2:	CD izquierda	0
bit 3:	Line derecha	1
bit 4:	Line izquierda	0
bit 5:	MIDI derecha	0
bit 6:	MIDI izquierda	0
bit 7:	- - -	0

Register 63: Left Input Gain Factor		
Nº bit	**Función**	**Valor por defecto**
bit 0-5:	- - -	0
bit 6:	Gain Factor	
bit 7:	Gain Factor	

El valor codificado en los dos bits es el factor por el que se desplaza el valor de entrada a la izquierda.

Ejemplo: bit 6: 0; bit7: 1 => Gain Factor = 4 (1 shl 2)

Registro 64: Right Input Gain Factor		
Nº bit	**Función**	**Valor por defecto**
bit 0-5:	- - -	0
bit 6:	Gain Factor	
bit 7:	Gain Factor	

Register 65: Left-Output Gain Factor		
Nº bit	**Función**	**Valor por defecto**
bit 0-5:	- - -	0
bit 6:	Gain Factor	
bit 7:	Gain Factor	

Register 66: Right Output Gain Factor		
Nº bit	**Función**	**Valor por defecto**
bit 0-5:	- - -	0
bit 6:	Gain Factor	
bit 7:	Gain Factor	

Register 67: AGC ON/OFF (= Automatic Gain Control)		
Nº bit	**Función**	**Valor por defecto**
bit 0:	Des/Activar AGC	0
bit 1-7:	- - -	0

Register 68: Left Treble		
Nº bit	**Función**	**Valor por defecto**
bit 0:	- - -	0
bit 1:	- - -	0
bit 2:	- - -	0
bit 3:	- - -	0
bit 4:	Treble Control	0
bit 5:	Treble Control	0
bit 6:	Treble Control	1
bit 7:	Treble Control	1

Register 69: Right Treble		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Treble Control	0
bit 5:	Treble Control	0
bit 6:	Treble Control	1
bit 7:	Treble Control	1

Register 70: Left Bass		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Bass Control	0
bit 5:	Bass Control	0
bit 6:	Bass Control	0
bit 7:	Bass Control	1

Register 71: Right Bass		
Nº bit	**Función**	**Valor por defecto**
bit 0:	---	0
bit 1:	---	0
bit 2:	---	0
bit 3:	---	0
bit 4:	Bass Control	0
bit 5:	Bass Control	0
bit 6:	Bass Control	0
bit 7:	Bass Control	1

13.2 Reproducción de archivos VOC con programación directa

Si quiere reproducir un archivo VOC mediante la Sound Blaster, puede emplear el controlador de Creative Labs. Creative Labs soporta este método, no publicando ningún tipo de información sobre la programación directa de la tarjeta Sound Blaster, sino sólo de sus controladores. Pero es más interesante no depender de un controlador, sino poder programar directamente la tarjeta. Por ello le presentamos aquí un pequeño VOC-Player, que controla la tarjeta directamente.

Las rutinas presentadas en este capítulo forman a la vez la base para el MOD-Player de la Sound Blaster, que desarrollaremos con usted en el apartado 15.2. El player emplea las rutinas presentadas en este apartado. Su código completo lo puede encontrar en el CD adjunto. Aquí no limitaremos a las rutinas principales para la reproducción VOC.

Comencemos con la rutina *Init_Voc*. Se le ha de pasar el nombre del archivo, si es necesario con vía de acceso, del archivo a reproducir. La rutina comprueba primero si ese archivo existe. A continuación se verifica la cabecera y finalmente se interpreta el primer bloque del archivo VOC. De este bloque la rutina obtiene las informaciones necesarias acerca de la tasa de muestreos y modo de reproducción (mono / estéreo). Ahora se puede alimentar un Double-Buffer con los datos del archivo, y la reproducción comienza mediante la salida directa del buffer actual a través de DMA. Esta "alimentación" de datos la realiza la interrupción SB.

Como puede ver no es muy difícil escribir un VOC-Player. Si lo desea puede añadir un tratamiento para otros bloques posibles en la interrupción. Normalmente los VOC suelen estar grabados, sin embargo, en una sola pieza, de modo que se pueden reproducir así.

```
procedure Init_Voc(filename : string);
const VOCkenn : string = 'Creative Voice File'+#$1A;
var ch : char;
    Cadena_Id : string;
    ct : byte;
    h : byte;
    error : integer;
    srlo,srhi : byte;
    SR : word;
    Samplingr : word;
    stereoreg : byte;
begin;
  Transfer_Testing := false;
  VOC_READY    := false;
```

```
vocsstereo := stereo;
stereo := false;

assign(vocf,filename);
reset(vocf,1);
if filesize(vocf) < 5000 then begin;
  VOC_READY   := true;
  exit;
end;
blockread(vocf,voch,$19);
Cadena_Id := voch.Cadena_Id;
if Cadena_Id <> VOCkenn then begin;
    { Identificador erróneo! }
    VOC_READY   := true;
    exit;
end;

  Blockread(vocf,inread,20);
  vblock.Identificador := inread[2];

if vblock.Identificador = 1 then begin;
  vblock.SR := inread[6];
end;

if vblock.Identificador = 8 then begin;
   SR := inread[6]+(inread[7]*256);
   Samplingr := 256000000 div (65536 - SR);
   if inread[9] = 1 then begin; {stereo}
     if sb16detected then samplingr := samplingr shr 1;
     stereo := true;
   end;
   vblock.SR := 256 - longint(1000000 DIV samplingr);
end;

if vblock.Identificador = 9 then begin;
   Samplingr := inread[6]+(inread[7]*256);
   if inread[11] = 2 then begin; {stereo}
     stereo := true;
     if sbprodetected then samplingr := samplingr * 2;
     vblock.SR := 256 - longint(1000000 DIV (samplingr));
  end else begin;
     vblock.SR := 256 - longint(1000000 DIV samplingr);
  end;
end;

  if vblock.SR < 130 then vblock.SR := 166;
  set_timeconst_sb16(vblock.SR);
```

```
    TamBloc := filesize(vocf) - 31;
    if TamBloc > 2500 then TamBloc := 2500;
    blockread(vocf,blk1^,TamBloc);

    ch := #0;
    fgr := filesize(vocf) - 32;
    fgr := fgr - TamBloc;
    Block_activ := 1;

  if fgr > 1 then begin;
    blockread(vocf,blk2^,TamBloc);
 fgr := fgr - TamBloc;
  end;

 Wr_dsp($D1);
 lastone := false;

 if not sb16Detected then begin;
   if Stereo then begin;
 stereoreg := Read_Mixer($0E);
 stereoreg := stereoreg OR 2;
 Write_Mixer($0E,stereoreg);
   end else begin;
 stereoreg := Read_Mixer($0E);
 stereoreg := stereoreg AND $FD;
 Write_Mixer($0E,stereoreg);
   end;
 end;

 Reprod_Bloque_DSP(TamBloc,blk1,false,true);
end;

procedure voc_done;
var h : byte;
begin;
 lastone := true;
 repeat until dsp_rdy_sb16;
 close(vocf);
 Reset_SBCard;
 stereo := vocsstereo;
end;
procedure dsp_int_sb16; interrupt;
{
 Este procedimiento es asaltado por una interrupción que se genera
 al final de una transferencia de bloques. Si no está activa la
 bandera Ultima_Reprod se inicia una nueva reproducción.
}
var h : byte;
```

```
begin;
  if interrupt_check then begin;
    IRQDetected := true;
  end else begin;
    if Transfer_Testing then begin;
      h := port[dsp_adr+$E];
      dsp_rdy_sb16 := true;

      if not Ultima_Reprod then begin;
       Reprod_Bloque_DSP(TamBlocoesse,blk,true,false);
      end;
    end else begin;
      h := port[dsp_adr+$E];
      if (fgr > TamBloc) and not lastone then begin
        lastone := false;
        if block_activ = 1 then begin
          Reprod_Bloque_DSP(TamBloc,blk2,false,true);
          blockread(vocf,blk1^,TamBloc);
          fgr := fgr - TamBloc;
          block_activ := 2;
        end else begin;
          Reprod_Bloque_DSP(TamBloc,blk1,false,true);
          blockread(vocf,blk2^,TamBloc);
          fgr := fgr - TamBloc;
          block_activ := 1;
        end;
      end else begin;
        if not lastone then begin;
          if block_activ = 1 then begin
           Reprod_Bloque_DSP(TamBloc,blk2,false,true);lastone :=true;
         end else begin;
           Reprod_Bloque_DSP(TamBloc,blk1,false,true);
            lastone := true;
         end;
        end else begin;
         dsp_rdy_sb16 := true;
         Wr_dsp($D0);
         VOC_READY := true;
        end;
      end;
    end;
  end;
  Port[$20] := $20;
end;
```

14. Gravis Ultrasound - la tarjeta de sonido de los fanáticos

La Gravis Ultrasound, llamada frecuentemente GUS por los expertos, es una tarjeta de sonido que emplea la nueva tecnología Wavetable. Destaca en que, a diferencia de la WaveBlaster de Creative Labs, los muestreos no se encuentran fijos en la ROM, sino que se cargan en la RAM de la Ultrasound. La RAM de la Ultrasound tiene un tamaño estándar de 256 KBytes, pero se puede ampliar hasta 1 MB en pasos de 256 KBytes. Esta ampliación es muy recomendable, y además bastante económica. Se emplean DRAMS, como también se pueden encontrar en una tarjeta VGA o en la placa madre de un 286. Estos se pueden emplear sin problemas, sólo hay que tener en cuenta la posición de los chips.

La Gravis Ultrasound es especialmente interesante por su característica de poder reproducir 32 canales simultáneamente. En este caso un canal designa una zona determinada en la DRAM de la Gravis. El chip de sonido de la Ultrasound reproduce los distintos muestreos con una velocidad "constante" de 44 kHz. Pero estos 44 kHz sólo se aplican a 14 voces. Para cada voz adicional se reduce la calidad de salida hasta 22 kHz para 28 voces. Si ahora reproduce un muestreo con una velocidad de 11 kHz, se generan huecos a causa de la velocidad de reproducción de 44 kHz. Estos huevos son llenados por el chip mediante una interpolación de los valores correspondientes lo que lleva a un aumento sustancial de la calidad.

La razón principal por la cual la Gravis Ultrasound es interesante para el programador, es el trabajo autónomo del chip. Así que una vez que haya cargado los datos de muestra en la DRAM de la tarjeta, puede iniciar los diferentes muestreos mediante una sencilla modificación de los parámetros de las voces individuales. Con ello, la tarjeta es muy adecuada para reproducir archivos MOD. A causa del poco tiempo que necesita la tarjeta para modificar los parámetros es especialmente interesante para programadores de juegos y demostraciones. Con ella se puede añadir un buen sonido incluso a las animaciones de cálculos intensivos.

14.1 Esta es la estructura de la Gravis Ultrasound

La Gravis Ultrasound se compone de tres unidades lógicas distintas. Al principio se encuentra el conector MIDI de la tarjeta. La interfaz MIDI 101 está integrada en

el chip GF1 de la tarjeta. Para todos los datos de entrada y salida se genera una interrupción. Esta interrupción se puede activar y desactivar mediante un registro de control. Para determinar el origen de las interrupciones se emplea un registro de estado. La siguiente unidad es un Eliminator Joystick Interface, desarrollado por Gravis. La interfaz del joystick se compone de un registro de ocho bits. Cuando se escribe un valor en el registro, se ponen a *0* cuatro Flip-Flops. Ahora, cuatro contadores registran el movimiento del joystick, relativo al momento del acceso de escritura. Si lee el registro de la interfaz, obtendrá secuencialmente el estado de los cuatro contadores y de los Flip-Flops. La interfaz se activa y desactiva mediante un jumper (puente) en la tarjeta.

El tercer y más importante de la Gravis Ultrasound es el chip GF1 desarrollado por Gravis. Este chip es muy polivalente. Con él puede reproducir datos de *8* y *16* bits, tanto *signed* como *unsigned*, en mono y en estéreo. Los datos se pueden transferir directamente o mediante transferencia DMA de *8* o *16* bits a y de la tarjeta.

El principio según el cual funciona la tarjeta, es la llamada tecnología Wavetable. Aquí se reproducen sonidos naturales (muestreos) de forma muy fiel. Mediante una modificación de la velocidad de reproducción se obtiene una modificación de la frecuencia del tono. Se puede generar otros efectos mediante la modulación de amplitud o frecuencia. La Ultrasound ofrece la posibilidad de regular independientemente para cada canal los parámetros de frecuencia de reproducción, modulación de amplitud y panning (balance de la voz).

14.2 Modo de funcionamiento de la GUS

Este apartado es interesante para todos aquellos que quieren saber cómo trabaja la Ultrasound a bajo nivel.

El GF1 es un así llamado procesador Pipeline. Constantemente recorre un bucle desde la voz 0 hasta el final de las voces definidas. Cada 1,6 microsegundos el chip realiza una serie de operaciones con las diferentes voces. Cuantas más voces haya definido, tanto más tarda el GF1 para procesarlas todas. Esto lleva a una calidad de reproducción descendente con un número creciente de voces. Puede terminar la frecuencia de la calidad de salida con la fórmula

```
Frec := 1.000.000 DIV (1.619695497 * Voces).
```

Así obtiene, por ejemplo para 20 voces, una frecuencia de salida de 30780 Hz. Debería realizar el cálculo de frecuencias, siempre que las necesite en sus programas, al principio y guardarlas en una tabla. Esto le ahorra el empleo de aritmética de coma flotante.

La interpolación de la GF1 es una razón de la calidad relativamente buena de la tarjeta. También se realiza con 16 bits para datos de 8 bits. Aclaremos la circunstancia de la interpolación con un ejemplo: Con 20 voces activas (= 30870 Hz) queremos reproducir un muestreo de 10 kHz. Es decir, que el GF1 realiza tres pasadas antes de alcanzar el siguiente bytes de datos del muestreo correspondiente (ya que 3 * 10 = 30). El GF1 calcula ahora el valor de la siguiente pasada según la fórmula

```
Valor de salida := (2/3 * Valor1 + 1/3 Valor2) DIV 2.
```

De forma análoga se forma el tercer valor de salida de 1/3 de *Valor1* y 2/3 de *Valor2*.

14.3 Registros de la GUS

La Ultrasound es direccionada por una multitud de registros, que parcialmente indexan otras subfunciones. La siguiente tabla ofrece un resumen de los registros. La *x* en las indicaciones de dirección se ha de sustituir por el valor base ajustado en cada caso. Si por ejemplo ha instalado la GUS con el port base *$240*, el registro de selección de voces del GF1 se encuentra en la dirección de port $342.

Dirección	Modo	Descripción
MIDI Interface		
$3x0	W	Control Register
$3x0	R	Status Register
$3x1	W	Enviar datos
$3x1	R	Recibir datos
Joystick Interface		
$201	W	Iniciar temporizador
$201	R	Registro de datos
GF1 Synthesizer		
$2x6	R	IRQ Status Register
$2x8	R/W	Timer Control Register
$2x9	W	Timer Daten Register
$3x2	R/W	Registro de selección de voces
$3x3	R/W	Registro de selección de funciones
$3x4	R/W	Registro de datos Lo Byte
$3x5	R/W	Registro de datos Hi Byte
$3x7	R/W	GUS DRAM
On Board		
$2x0	W	Mixer Control Register
$2xB	W	IRQ Control Register
$2xB	W	DMA Control Register

Como ya hemos mencionado anteriormente, la GUS se compone de diferentes unidades de interfaz. Para un músico es extremadamente interesante la interfaz MIDI. Se encuentra en las direcciones *$3x0* y *$3x1*. Consideremos primero la dirección *$3x0*:

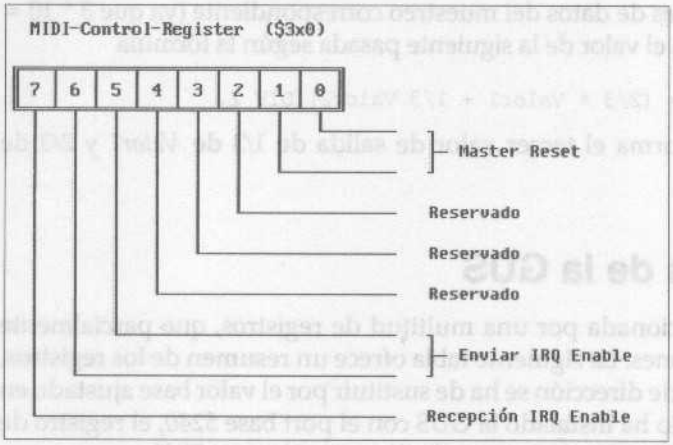

Figura 25: El registro de control MIDI

Cuando escriba en esta dirección, está direccionando con ello el registro de control MIDI. Para realizar un Master-Reset, ha de colocar los bits 0 y 1 primero a *1* y después de un pequeño retardo a *0*. Han de permanecer en el valor *0*, para garantizar un trabajo sin problemas. Para activar la interrupción de transferencia, ha de poner a *1* el bit 5 y a *0* el bit 6. La interrupción de recepción se activa mediante un *1* en el bit 7. Cuando lea en la dirección *$3x0*, obtendrá los datos del registro de estado MIDI.

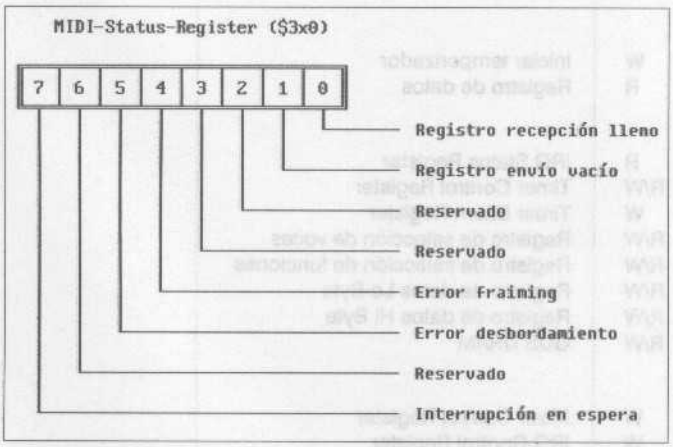

Figura 26: El registro de estado MIDI

Puede encontrar la explicación de la ocupación de los bits del registro en la figura. El registro *$3x0* se comporta de forma idéntica a un *6850 UART*, la interfaz estándar MIDI.

En la dirección $3x1 se encuentra el registro de datos MIDI. Tiene una anchura de 8 bits y puede ser leído y escrito. En él puede encontrar los datos que llegan de un dispositivo MIDI. Si quiere enviar datos, ha de enviarlos a esta dirección de port.

De interés para programadores de juegos debería ser la interfaz de joystick de Advanced Gravis. Se encuentra en la dirección $201. Si escribe un valor en esta dirección, se ponen a 0 los cuatro contadores de la interfaz comenzando a registrar los movimientos del joystick. Mediante una lectura del port se pueden consultar secuencialmente los cuatro contadores y el estado del Flip-Flop correspondiente.

La interfaz más importante la representa la interfaz GF1. GF1 puede gestionar hasta 32 voces. Para no tener que indicar el número de canal a modificar en todas las operaciones, existe el registro de selección de voces. Este se encuentra en la dirección $3x2 y tiene una anchura de 8 bits. Puede ser leído y escrito y se permiten valores entre 0 y 31. Un número mayor puede provocar un funcionamiento erróneo de la tarjeta..

Un registro muy importante es el registro de selección de funciones. Tiene una anchura de 8 bits y pone a disposición toda una serie de subfunciones. Puede ser leído y escrito. Mediante la dirección $3x3 primero ha de seleccionar la función correspondiente del registro a encontrar, y escribir o leer entonces los datos del port de datos correspondiente. Veamos primero las funciones independientes de voz del registro de selección de funciones:

Función	Modo	Anchura	Descripción
$41	R/W	8	DRAM DMA Control
$42	W	16	Dirección inicio DMA
$43	W	16	Dirección LO DRAM I/O
$44	W	8	Dirección Hi DRAM I/O
$45	R/W	8	Timer Control Register
$46	W	8	Contador Timer 1
$47	W	8	Contador Timer 2
$48	W	8	Frecuencia de muestreo
$49	R/W	8	Sampling Control
$4B	W	8	Ajuste de Joystick
$4C	R/W	8	Reset

La primera función se encuentra bajo el número de índice $41. Designa el registro de control DRAM-DMA. El registro tiene el siguiente aspecto:

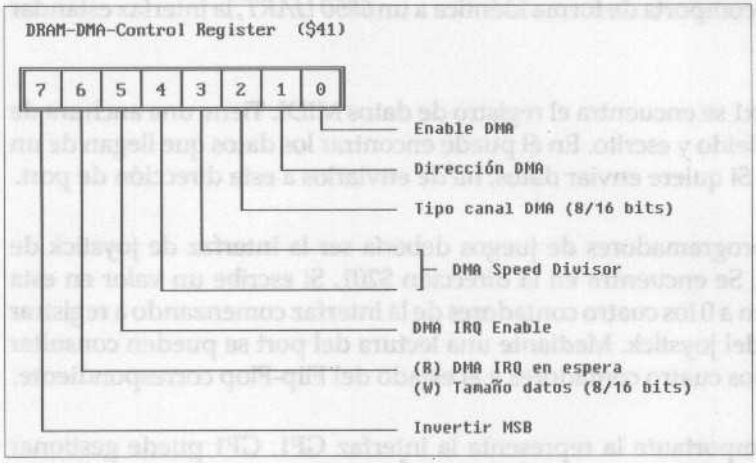

Figura 27: El registro de control DRAM-DMA

El registro controla la transferencia de datos entre la memoria principal y la Gravis Ultrasound vía DMA. Para iniciar una transferencia, ha de poner a 1 el bit 0. Con el bit 1 determina la dirección de la transferencia. Si tiene el valor 0, se transfieren datos de la memoria principal a la DRAM de la GUS, con un valor de 1 la transferencia va desde la GUS a la memoria principal. El tipo de canal empleado se determina en el bit 2. Si aquí se encuentra un 0, se emplea un canal de 8 bits (0-3), un 1 designa un canal de 16 bits (4-7). En los dos bits 3 y 4 se encuentra el valor del divisor DMA-Speed.

La velocidad máxima se encuentra aproximadamente en 650 kHz, se divide entre el divisor, que puede tener un valor entre 1 y 4. Con el bit 5 determina si se ha de disparar una interrupción cuando se finaliza la transferencia de bloque. Un valor 1 activa la interrupción, 0 la desactiva. El bit 6 tiene diferentes significado, según si envía datos a la GUS o los recibe de ella.

Si se trata de un acceso de lectura, el bit 6 determina si la interrupción DMA se ha de conmutar a espera. Durante una transferencia a la Gravis Ultrasound, se ajusta el tamaño de los datos enviados. El valor 0 indica datos de 8 bits y 1 los de 16 bits. El tamaño de los datos es independiente del tipo de canal DMA determinado. El bit 7 finalmente indica si se ha de invertir el bit alto, para convertir los datos en un complemento a dos. El bit alto es el bit 7 en datos de 8 bits, y el bit 15 en datos de 16 bits.

Con la función $42 (dirección de inicio DMA) determina la dirección de inicio para una transferencia DMA a/de la RAM de la GUS. Si quiere transferir datos de 8 bits, estos han de comenzar en una dirección divisible entre 16, en el caso de

datos de 16 bits en una dirección divisible entre 32, Si esto no es así, la GUS redondea hasta la siguiente dirección divisible. La razón de esta limitación es que sólo se emplean 16 bits de dirección, aunque son necesarios 19 bits para direccionar 1 MByte. Por ello los bits 15 hasta 0 corresponden a las líneas de direcciones 19 a 4.

Mediante los registros de direcciones DRAM-I/O $43 y $44 puede ajustar la dirección de la DRAM en/de la que puede /escribir/leer datos directamente. En el registro $43 se fijan las 16 líneas de direcciones inferiores, en $44 se escriben las cuatro líneas de direcciones superiores (bits 0 hasta 3). Cuando haya determinado una dirección, puede leer o escribir la posición de memoria mediante el registro GUS-DRAM ($3x7). El registro de control del temporizador ($45) se necesita para activar o apagar el temporizador integrado. Mediante el bit 2 se activa la interrupción del temporizador 1; el bit 3 es responsable del temporizador 2.

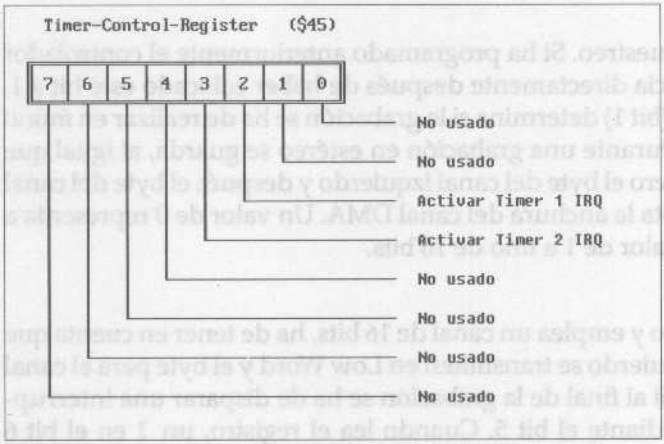

Figura 28: El registro de control del temporizador

Los registros de contador del temporizador 1 y 2 se encuentran como función $46 y $47. Los contadores se han de cargar con el valor inicial y cuentan entonces hasta $FF. Al alcanzar este límite, generan una interrupción. El temporizador 1 se incrementa cada 80 microsegundos y el temporizador cada 320 microsegundos. Con el registro de frecuencia de muestreo ($48) puede ajustar la velocidad de grabación de la tarjeta. El valor de 8 bits se calcula según la fórmula

```
SRQ := 9878400 DIV ((Frecuencia + 2) SHL 4)
```

El registro de control de muestreo ($49) es importante. Con él comienza la grabación mediante DMA.

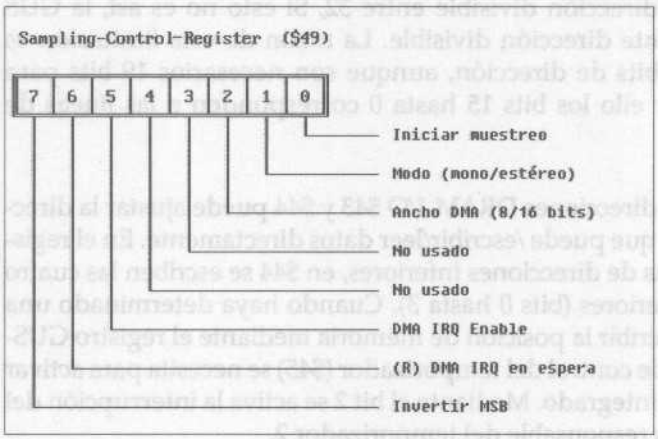

Figura 29: El registro de control de muestreo

Con el bit 0 se inicia el muestreo. Si ha programado anteriormente el controlador DMA, la grabación se inicia directamente después de haber colocado este bit a 1. Mediante el bit de modo (bit 1) determina si la grabación se ha de realizar en mono (=0) o en estéreo (=1). Durante una grabación en estéreo se guarda, al igual que en la Sound Blaster, primero el byte del canal izquierdo y después el byte del canal derecho. Con el bit 2 ajusta la anchura del canal DMA. Un valor de 0 representa a un canal de 8 bits, y un valor de 1 a uno de 16 bits.

Si quiere grabar en estéreo y emplea un canal de 16 bits, ha de tener en cuenta que los bytes para el canal izquierdo se transmiten en Low Word y el byte para el canal derecho en High Word. Si al final de la grabación se ha de disparar una interrupción, puede activarla mediante el bit 5. Cuando lea el registro, un *1* en el bit 6 indica que hay una interrupción en espera. Mediante el bit 7 finalmente puede activar la inversión del High-Bit anteriormente descrita.

La función *$4B* sirve para el ajuste del Joystick. A ser posible no debería realizar modificaciones en los ajustes, simplemente los mencionamos como dato adicional. Mediante un valor de *$29* se ajusta el registro con un valor de *4,3 voltios*. Un valor superior o inferior sólo es necesario si quiere emplear un ordenador muy lento o muy rápido. Es preferible emplear la utilidad *Ultrajoy* suministrada por Gravis para ajustar este registro. De nuevo, como importante, se podría calificar el registro de reset *$4C*.

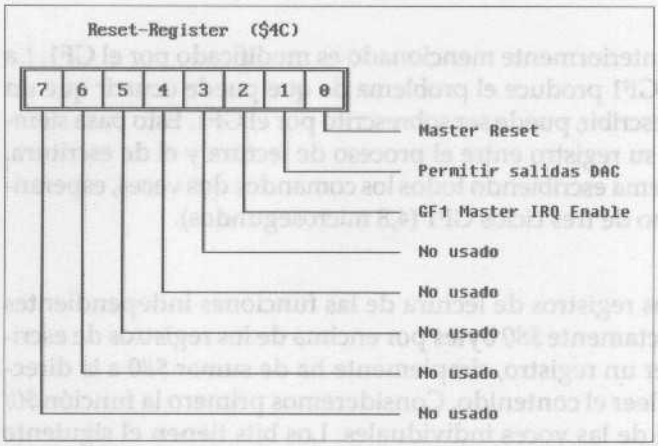

Figura 30: El registro de reset

El registro normalmente contiene el valor *$07*. Mediante el bit 0 puede provocar un Master-Reset (reinicialización general) del GF1. El valor *1* representa el funcionamiento normal y *0* significa un reset. El estado de reset permanece activo mientras no se vuelva a poner a *1* el valor. Mediante el bit 1 puede activar la reproducción del DAC. Así que si detiene todas las voces de su programa, pero no quiere interrumpir la reproducción, solo necesita poner a *0* el bit 1. El bit 2 ha de valer *1* si quiere emplear las interrupciones del la GUS.

Después de haber conocido las funciones independientes de voz, vamos a describir las dependientes de voz. Para comenzar, un resumen de las funciones disponibles:

Escribir	Leer	Anchura	Descripción
$00	$80	8	Control de voz
$01	$81	16	Control de frecuencia
$02	$82	16	Dirección inicial Hi
$03	$83	16	Dirección inicial Lo
$04	$84	16	Dirección final Hi
$05	$85	16	Dirección final Lo
$06	$86	8	Volume Ramp Rate
$07	$87	8	Volume Ramp Inicio
$08	$88	8	Volume Ramp Fin
$09	$89	16	Volumen actual
$0A	$8A	16	Dirección actual Hi
$0B	$8B	16	Dirección actual Lo
$0C	$8C	8	Pan Position
$0D	$8D	8	Control de volumen
$0E	$8E	8	Voces activas (independiente de voz)
$0E	$8F	8	Fuente IRQ (independiente de voz)

Algunos de los registros anteriormente mencionado es modificado por el GF1. La arquitectura Pipeline del GF1 produce el problema de que puede ocurrir que un valor que usted acaba de escribir, puede ser sobrescrito por el GF1. Esto pasa siempre que si ha modificado su registro entre el proceso de lectura y el de escritura. Puede solucionar el problema escribiendo todos los comandos dos veces, esperando entre ambos un mínimo de tres ciclos GF1 (4,8 microsegundos).

Otra propiedad es que los registros de lectura de las funciones independientes de voz se encuentran exactamente *$80* bytes por encima de los registros de escritura. Así que si quiere leer un registro, simplemente ha de sumar *$80* a la dirección de escritura y podrá leer el contenido. Consideremos primero la función *$0/$80*. Sirve para el control de las voces individuales. Los bits tienen el siguiente significado.

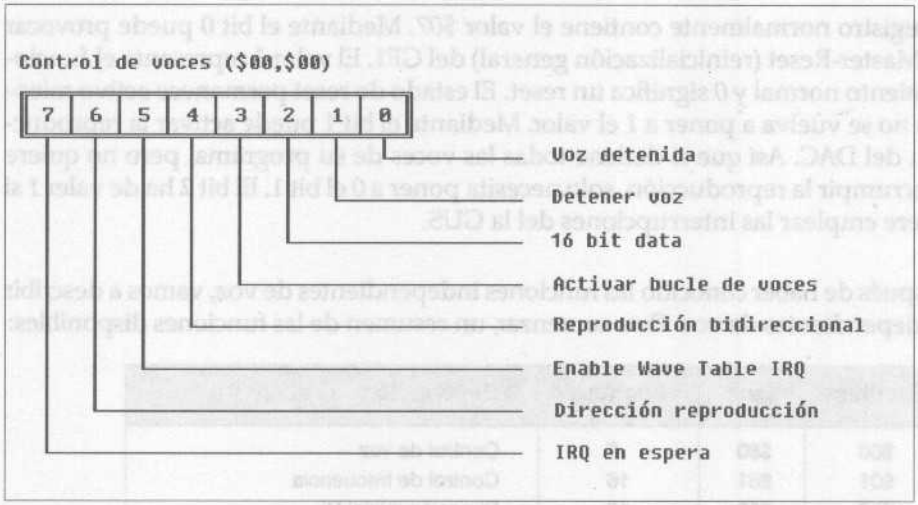

Figura 31: Control de voces

Los bits 0, 6 y 7 son modificados por el GF1. Si tienen algún significado para usted, debería realizar un doble acceso. El bit 0 indica si la voz se reproduce, o si ha alcanzado la posición final. *0* representa una voz activa. Con el bit 1 puede detener la voz. Esta es una propiedad muy importante, por ejemplo para un MOD-Player. El bit 2 indica si se emplean datos de 8 bits o de 16. Un *1* indica datos de 16 bits y un *0* datos de 8 bits. Con el bit 3 puede activar y desactivar el Looping de una voz. Esto es muy útil, ya que en la rutina de reproducción ya no se ha de ocupar por un looping correcto. Si pone el bit 4 a *1*, obtendrá un Looping bidireccional. Es decir, la voz se reproduce primero hacia adelante, después hacia atrás, después de nue-

vo hacia adelante, etc. En el Looping normal, la voz comienza de nuevo desde el principio cuando llega al final y sólo se reproduce en un sentido. A través del bit 5 puede activar la IRQ de la Wavetable. Si la voz alcanza su posición final, se genera una interrupción. Esta, por cierto, también se genera si el Looping está activado. El bit 6 indica la dirección de reproducción. Un *0* indica direcciones ascendentes (reproducción hacia adelante) y un *1* representa direcciones descendentes. Finalmente, el bit 7 indica que hay una IRQ en espera. Cuando se alcanza el final de la voz, se van generando interrupciones en el modo de no repetición (Looping) hasta que se detenga la voz.

Con el registro de control de frecuencias ($01/$81) puede determinar con qué velocidad se ha de reproducir la voz. En los bits 15 hasta 10 se encuentra la mantisa del factor por el cual avanza la voz. Los bits 9 hasta 1 contienen la parte decimal. El bit 0 no se emplea. Si por ejemplo quiere reproducir una voz con 22 kHz, debería proceder según el siguiente método:

Voces	Divisor
14	43
15	40
16	37
17	35
18	33
19	31
20	30
21	28
22	27
23	26
24	25
25	24
26	23
27	22
28	21
29	20
30	20
31	19
32	18

Divida la frecuencia (22000) entre el divisor (por ejemplo 30 con 20 voces). El resultado se ha de enviar al port $3x5. Con la pareja de registros ($02/$82) y ($03/$83) se fija la dirección de inicio de la voz en la GUS-DRAM (líneas de dirección 19-7). Los bits 15 hasta 13 de este registro no se emplean. Los 7 bits inferiores de la dirección inicial se ajustan mediante los bits 15 hasta 9 de la pareja de registros $03/$83. Así que corresponden a las líneas de direcciones 6 hasta 0. Los bits 8

hasta 5 del registro indican la posición decimal de la dirección inicial y no se emplean. La dirección final de una voz se pueden ajustar mediante las parejas de registro $04/$84 y $05/$85. La ocupación de los bits es análoga a la de la dirección inicial.

La Gravis Ultrasound soporta el Volume Sliding por hardware. Con la pareja de registros $6/$86 puede ajustar la tasa del Volume-Ramping. Los 6 bits inferiores designan el tamaño del paso en el que se incremente o disminuye el volumen. Un 1 es el incremento más lento del mismo y un 63 el mayor valor posible. Los dos bits superiores designan la velocidad del Ramping. Se aplica lo siguiente:

Bit 7 y 6	Tasa
00	1 DIV (1,6 * voces activas)
01	8 DIV (1,6 * voces activas)
10	64 DIV (1,6 * voces activas)
11	512 DIV (1,6 * voces activas)

El Ramping más rápido se obtiene con una tasa de *00b* y un paso de *63*. El más lento con una tasa de *11b* y un paso de *1*.

Mediante las parejas de registro $7/$87 y $8/$88 se ajusta el volumen inicial y final del Volume-Ramping. Los bits 7 hasta 4 representan el exponente del valor del volumen y la base se forma por los bits 3 hasta 0. Debería borrar los 4 bits inferiores del volumen original, para alcanzar el valor de Ramping. Tenga en cuenta que el volumen inicial siempre ha de ser menor que el volumen final. Para un Ramping down ha de fijar el bit de dirección en el registro de control de volumen para reducción.

El volumen actual se ajusta mediante la pareja de registros *$09/$89*. En los bits 15 hasta 12 se encuentra en exponente del volumen y en los bits 11 hasta 4 la mantisa. El valor ampliado para la mantisa se necesita para una mayor precisión durante el Volume-Ramping. Los bits 3 hasta 0 del registro no se emplean. Este registro se ha de igualar con el valor inicial del Volume-Ramping.

Con las parejas de registro $0A/$8A y $0B/$8B tiene acceso a la posición actual de la voz seleccionada. La ocupación de bits corresponde a la del registro $02/$82, la dirección inicial.

Puede ajustar la posición del panning mediante el registro *$0C/$8C*. Los bits 7 hasta 4 no se emplean, en los bits 3 hasta 0 se encuentra la posición. El valor *0* representa la posición más a la izquierda, *15* la situada más a la derecha. La pareja de

registros *$0D/$8D* es muy importante para el Volume-Ramping. Se trata del registro del control de volumen.

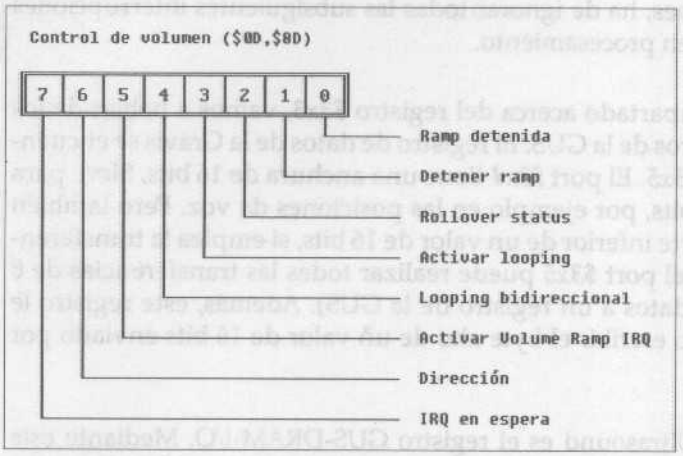

Figura 32: El registro del Volume-Ramping

Los bits 0, 6 y son modificados por el GF1. Así que también aquí ha de emplear el acceso doble. El bit 0 muestra que el Ramping ha terminado. Con el bit 1 puede detener el Ramping. Si lo pone a *1*, el Ramping se detiene inmediatamente. El bit 2 sirve como identificador para el "Rollover". Más adelante explicaremos qué es. Mediante el bit 3 puede activar *(=1)* y desactivar el Looping del Ramping. El bit 4 determina si el Ramping sólo se realiza en una dirección *(=0)* o de forma bidireccional. Mediante la activación del bit 5 dispara una interrupción al alcanzar el final del Volume-Ramp. El bit 6 determina la dirección del Volume-Ramp. Un valor de *0* indica un volumen creciente y *1* un volumen decreciente. El bit 7 indica si hay en espera una IRQ de Volume-Ramp.

El registro *$0E/$8E* es independiente de voz. Con él puede determinar la cantidad de voces activas. Los bits 7 a 5 siempre han de contener el valor *1*, los bits 4 hasta 0 contienen la cantidad de voces empleadas *-1*. Por favor tenga en cuenta que el número mínimo de voces es *14*.

Mediante el registro fuente de IRQ puede determinar la razón por la cual se dispara una IRQ de Wavetable. Los bits 4 hasta 0 contienen el número de la voz a disparar *-1*. El bit 5 siempre tiene el valor *1*. Si el bit 6 vale *0*, hay en espera una Volume-Ramp-IRQ. Un valor *0* en el bit 7 indica que hay en espera una Wavetable-IRQ. En este registro es importante que sus valores se pierdan cada vez que sea leído.

En su controlador de interrupciones ha de procesar todos las IRQ apuntadas, es decir, las de Volume-Ramp y Wavetable de la voz correspondiente. Ya que teóricamente es posible que dos voces disparen simultáneamente una interrupción, ha de procesar los registros hasta que tanto el bit 6 como el bit 7 estén a *1*. Ya que una voz podría volver a disparar una interrupción, mientras está procesando el controlador de interrupciones, ha de ignorar todas las subsiguientes interrupciones de una voz que ya está en procesamiento.

Después de este largo apartado acerca del registro *$3x3*, vamos a hablar de los no tan complejos registros de la GUS. El registro de datos de la Gravis se encuentra en los ports *$3x4* y *$3x5*. El port *$3x4* tiene una anchura de 16 bits. Sirve para acoger un valor de 16 bits, por ejemplo en las posiciones de voz. Pero también puede reproducir la parte inferior de un valor de 16 bits, si emplea la transferencia de 8 bits. Mediante el port *$3x5* puede realizar todas las transferencias de 8 bits (transferencias de datos a un registro de la GUS). Además, este registro le puede servir para leer o escribir el byte alto de un valor de 16 bits enviado por *$3x4/$3x5*.

El registro *$3x7* de la Ultrasound es el registro GUS-DRAM-I/O. Mediante este registro puede escribir o leer datos directamente de/a la RAM de la GUS. Tenga en cuenta que anteriormente ha de ajustar primero la dirección DRAM de la transferencia mediante las funciones *$43* y *$44*.

Después de haber conocido los registro *3x?* de la GUS queremos presentar ahora los registros *2x?* de la Gravis. El más elemental se encuentra en la dirección *$2x0*, la dirección base de la Gravis Ultrasound. Es el registro de control de mezcla (Mixer-Control-Register). Tiene la siguiente estructura:

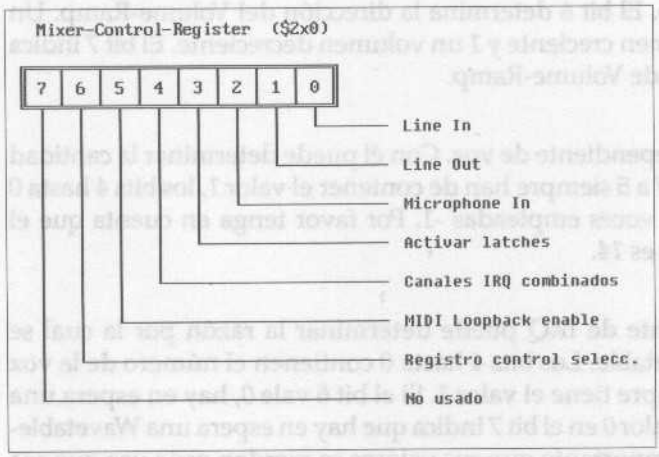

Figura 33: El registro de control de mezcla

Mediante el borrado del bit 0 activa la señal de Line-IN de la GUS. Análogamente, un borrado del bit 1 activa la señal Line-OUT. Si además quiere activar el micrófono de la GUS, ha de poner a *1* el bit 2. El bit 3 es muy importante. Mediante él puede activar las interrupciones de la tarjeta y los canales DMA. Una vez que los haya conectado, no debería desconectarlos, ya que de lo contrario podría obtener IRQ aleatorias. El bit 4 indica si el canal IRQ *1*, que es responsable para el GF1, se ha de combinar con el canal IRQ 2 (MIDI). Mediante el bit 5 activa el MIDI-Loopback. Esto significa que todos los datos MIDI de salida se vuelven a enviar al port de entrada MIDI. El bit 6 es como una protección contra aplicaciones que buscan tarjetas salvajemente. Si el bit se pone a *0*, el siguiente acceso al port *$2xB* modifica los Latches DMA, con un valor de *1* se modifican los Latches IRQ. El acceso al port *$2xB* ha de ser el siguiente acceso al port. Sino se ignora el acceso a los Latches. Esto evita que los Latches se modifiquen involuntariamente por los valores de prueba de otros programas. El registro de estado de IRQ *$2x6* está en estrecha relación al registro fuente IRQ. Mediante el registro de estado de IRQ puede determinar qué unidad de la Ultrasound ha disparado la interrupción.

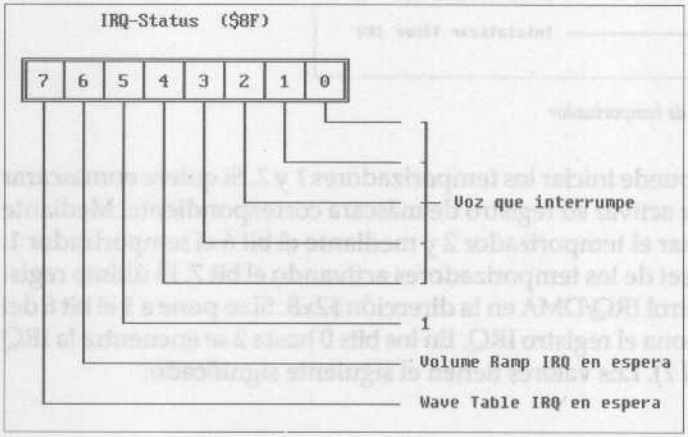

Figura 34: El registro de estado de IRQ

El causante de las interrupciones de la Ultrasound se puede determinar según qué bit esté activo en el registro. El bit *0* representa el envío de datos MIDI, el bit *1* la recepción de datos MIDI. Si el bit 2 está activo, la interrupción fue disparada por el temporizador 1, con el bit 3 el causante fue el temporizador 2. El bit 4 no se emplea. El bit 5 indica una Wavetable-Interrupt, el bit 6 una Volume-Ramp-Interrupt. Tenga en cuenta que con ambos IRQ ha de determinar la voz que produce el disparo. El bit 7 se activa cuando la interrupción se disparó a causa del fin de una transferencia DMA.

El registro *$2x8* es el registro de control de temporizador compatible Adlib. Mediante la escritura del valor *4* puede activar los temporizadores. El bit 6 se activa cuando el temporizador *1* ha finalizado. En el caso en que el temporizador haya cumplido, se pone a *1* el bit 5. El registro *$2x9* es el registro de datos de temporizador. Mediante este registro puede controlar los temporizadores propios de la GUS.

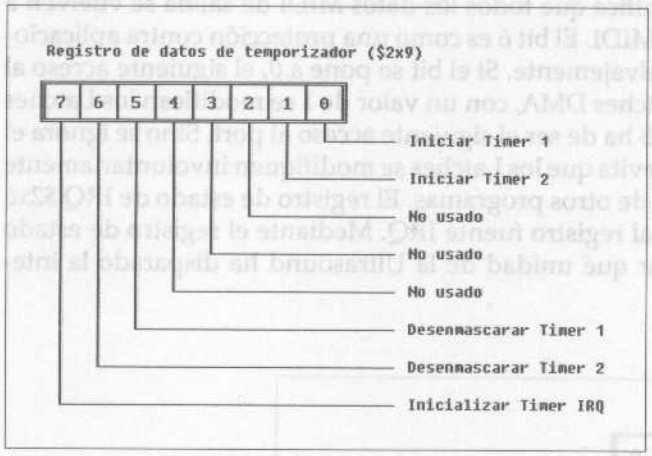

Figura 35: El registro de datos de temporizador

Mediante los bits 0 y 1 puede iniciar los temporizadores *1* y 2. Si quiere enmascarar un temporizador, ha de activar su registro de máscara correspondiente. Mediante el bit 5 puede direccionar el temporizador 2 y mediante el bit 6 el temporizador 1. Puede conseguir un reset de los temporizadores activando el bit 7. El último registro es el registro de control IRQ/DMA en la dirección *$2xB*. Si se pone a 1 el bit 6 del registro *$2x0*, se direcciona el registro IRQ. En los bits 0 hasta 2 se encuentra la IRQ para el chip GF1 (*Canal 1*). Los valores tienen el siguiente significado:

0	Sin interrupciones
1	IRQ 2
2	IRQ 5
3	IRQ 3
4	IRQ 7
5	IRQ 11
6	IRQ 12
7	IRQ 15

Mediante los bits 3 hasta 5 puede elegir la IRQ para el canal MIDI (*Canal 2*). Se emplean las siguientes correspondencias:

0	Sin interrupciones
1	IRQ 2
2	IRQ 5
3	IRQ 3
4	IRQ 7
5	IRQ 11
6	IRQ 12
7	IRQ 15

Si el bit 6 está a *1*, esto significa que se combinan las dos IRQ, es decir, que comparten el mismo canal. En este caso se utiliza la IRQ del GF1 (*Canal 1*). El segundo canal se ha de poner a *0*, ya que de lo contrario se crearía un conflicto de bus. El bit 7 no se emplea. Si modifica los ajustes para las IRQ, esto normalmente lleva a una interrupción en el canal antiguo. Esto es debido a que las IRQ ya no son controladas por los Latches. Pero esto no debería representar ningún problema. Si el bit 6 del port *$2x0* tiene el valor *0*, se direcciona el registro de control DMA. Mediante los bits 0 hasta 2 de este registro determina el primer canal DMA. Se emplea lo siguiente:

0	Sin canal DMA
1	Canal DMA 1
2	Canal DMA 3
3	Canal DMA 5
4	Canal DMA 6
5	Canal DMA 7
6	no permitido
7	no permitido

Mediante los bits 3 hasta 5 puede seleccionar el segundo canal DMA. También aquí se aplica:

0	Sin canal DMA
1	Canal DMA 1
2	Canal DMA 3
3	Canal DMA 5
4	Canal DMA 6
5	Canal DMA 7
6	no permitido
7	no permitido

Un bit 6 activado indica que tanto el GF1 como el MIDI emplean ambos el mismo canal. Si los dos canales van a través del mismo canal DMA, se ha de escribir el valor *0* para el segundo canal DMA, ya que de lo contrario aparecerían conflictos de bus. El bit 7 no se emplea.

14.4 Programación de la GUS

Después de haber conocido el hardware de la GUS, vamos a explicar ahora la programación de la tarjeta. Aunque las funciones básicas de la misma son bastante fáciles de programar, ésta tiene sus pequeños trucos. No se sorprenda por ejemplo de que sus rutinas de temporizador no funcionan si no ha permitido las interrupciones globales. Pero no tema, iremos desarrollando todas las rutinas necesarias y fundamentales paso a paso con usted. En el apartado 15.3 puede encontrar un ejemplo de aplicación para su Gravis: un MOD-Player que puede integrar sin problemas en sus propios programas.

Así se inicializa la GUS

Las rutinas básicas aquí representadas para la GUS están todas en ensamblador. No debería representar ninguna dificultad para usted el emplearlas junto con cualquiera de los lenguajes corrientes. Es muy aconsejable realizar las rutinas en ensamblador, ya que sólo a nivel de ensamblador dispone del control total necesario sobre la máquina.

Comencemos con una rutina fundamental, que no puede faltar en ningún programa GUS: la inicialización de la tarjeta. ¿Cuál es la estructura de un procedimiento de inicialización para la Gravis?

Para comenzar utilizaremos naturalmente la función de reset de la tarjeta. Pero esto no es todo. El registro de control DMA, el registro de control de temporizadores y el registro de control de muestreo se han de reinicializar "a mano". Además, el procedimiento de inicialización debería determinar el número de voces empleadas, determinar los parámetros de los diferentes canales y colocar el volumen de las diferentes voces a *0*. Con ello el procedimiento se hace un poco más largo, pero así se evitan posibles y desagradables efectos secundarios como por ejemplo un súbito e indefinible graznido desde la izquierda, etc.

Puede obtener una inicialización de la tarjeta por hardware, seleccionando mediante el registro de comandos la función *Init* y colocándola a *0*. Espere algunos ciclos GF1, y vuelva a colocar el registro a *1*, sino la tarjeta permanecerá en el estado de reset y no podrá ser empleada correctamente.

A continuación ha de reiniciar los registros de control DMA, de control de los temporizadores y de control del muestreo. En la práctica todos los tres registros deberían tener su estado por defecto cuando contienen el valor *0*. Así que simplemente se han de seleccionar mediante el registro de comandos y ser escritos con *0*. Mediante la función *0Eh* del registro de comandos se puede seleccionar entonces el registro de voces. tenga en cuenta que los dos bits superiores del byte con el

número de voces han de estar a *1*. La manera más sencilla de conseguir esto es cargando el byte en uno de los registros y cargándolo allí con el valor correcto mediante un *OR 0C0h*.

Si ya ha instalado su controlador de interrupciones (esto no es necesario para el funcionamiento correcto de la GUS), también se habrá de encargar de las interrupciones durante el reset. Vacíe primero las posibles interrupciones DMA en espera, leyendo primero el registro de estado y a continuación el registro de control de DMA. Mediante un acceso de lectura al registro de control de muestreo debe borrar a continuación las interrupciones pendientes de la Wavetable. Estas se pueden omitir simplemente, ya que no tienen ningún significado para nosotros. ¡Al fin y al cabo estamos reinicializando la tarjeta!

Como siguiente paso debería reiniciar en un bucle todos los canales. Es interesante el reponer todos los 32 canales y no sólo los que se van a necesitar, más que nada para evitar pposibles efectos indeseados. Para ello seleccione en el bucle la voz a procesar, y fije el modo para la misma en *3*. Con ello detendrá eventualmente la reproducción de las voces. A continuación debería colocar el volumen a *0* mediante el registro de volumen, También puede inicializar los datos de looping de la voz, aunque esto no tiene mucho sentido, ya que estos se han de fijar de todos modos antes de poder emplear la voz.

Después de haber inicializado las voces en el bucle, debería procesar las posibles interrupciones que hayan podido aparecer mientras. Como último paso realice un reset por hardware mediante el registro de reset y active las interrupciones así como la reproducción de muestreos en el registro GF1-Master. Puede encontrar un ejemplo de función de reset en el procedimiento de ensamblador *U_Inicializa*.

```
u_Inicializa proc near
; ##################################################################
;   ### inicializa la Ultrasound                                 ###
; ##################################################################
  mov bx,w u_Command
  mov cx,w u_datahi
  mov dx,bx
  mov al,4ch                    ; seleccionar registro Init
  out dx,al
  mov dx,cx
  mov al,0                      ; realizar init
  out dx,al
  call u_delay                  ; esperar
  call u_delay
  mov dx,bx
  mov al,4ch
  out dx,al
```

```
mov dx,cx
mov al,1                        ; terminar init
out dx,al
call u_delay
call u_delay
mov dx,bx                       ; reset DMA Control Register
mov al,41h
out dx,al
mov dx,cx
mov al,0
out dx,al
mov dx,bx                       ; reset Timer Control Register
mov al,45h
out dx,al
mov dx,cx
mov al,0
out dx,al
mov dx,bx                       ; reset Sampling Control Register
mov al,49h
out dx,al
mov dx,cx
mov al,0
out dx,al
mov dx,bx                       ; fijar nº voces
mov al,0Eh
out dx,al
add dx,2
mov al,Num_Voices
or  al,0C0h
out dx,al
mov dx,w u_status               ; vaciar posibles interr. DMA
in al,dx
mov dx,bx
mov al,41h
out dx,al
mov dx,cx
in al,dx
mov dx,bx                       ; vaciar Sampling Interrupts
mov al,49h
out dx,al
mov dx,cx
in al,dx
mov dx,bx                       ; leer registro IRQ Status
mov al,8Fh                      ; => no quedan interr. sin procesar
out dx,al
mov dx,cx
in al,dx
push bx                         ; apagar las voces en bucle
push cx
```

```
    mov cx,0
  @VoiceClearLoop:
    mov dx,w u_Voice                ; seleccionar voz
    mov al,cl
    out dx,al
    inc dx
    mov al,0                        ; fijar Voice Mode
    out dx,al
    add dx,2
    mov al,3                        ; parar voz
    out dx,al
    sub dx,2                        ; volumen a 0
    mov al,0dh
    out dx,al
    add dx,2
    mov al,3
    out dx,al
    inc cx
    cmp cx,32                       ; repetir para todas las voces
    jnz @VoiceClearLoop
    pop cx
    pop bx
    mov dx,bx                       ; procesar eventuales interr.
    mov al,41h
    out dx,al
    mov dx,cx
    in al,dx
    mov dx,bx
    mov al,49h
    out dx,al
    mov dx,cx
    in al,dx
    mov dx,bx
    mov al,8fh
    out dx,al
    mov dx,cx
    in al,dx
    mov dx,bx                       ; realizar reset
    mov al,4ch
    out dx,al
    mov dx,cx                       ; activar GF1 Master IRQ
    mov al,7
    out dx,al
    ret
  u_Inicializa endp
```

Pero antes de poder realizar un reset, ha de saber en qué port se encuentra la Gravis. Existe la posibilidad de dejar que el usuario introduzca los parámetros para la GUS. Esta posibilidad es más cómoda para el programador, pero no dice

demasiado sobre nuestros conocimientos, además de no ser precisamente cómoda para el usuario. Es mucho más adecuado hacerlo mediante una rutina de test, de fácil realización, en la que se reconoce el port base de la Gravis.

Si quiere desarrollar semejante rutina de test, debería proceder de la siguiente forma: Compruebe en un bucle de *10h* todos los ports posibles entre *200h* y *280h*. Inicialice a la vez la tarjeta, escriba datos a la RAM de la GUS y vuelva a leer estos datos. Si coinciden, ha encontrado el port base de la GUS. Si no, ha de probar el siguiente port, o no existe ninguna Gravis Ultrasound.

Puede encontrar un ejemplo de este tipo de rutina de detección en el procedimiento *Detect_Gus*. El procedimiento inicializa primeramente la GUS mediante el registro de inicialización. A continuación se escribe un byte mediante la función *Poke* en la DRAM de la GUS. Naturalmente sólo se estará escribiendo en la DRAM de la tarjeta si se eligió el port base correcto. De lo contrario, el comando *out* "desaparece" sin ningún efecto. A continuación espere un tiempo adecuado para estar completamente seguro de que el GF1 no se meta en medio. Como último paso puede leer ahora el byte de la RAM de la GUS mediante un comando *Peek*.

Si el valor leído coincide con el escrito, ha encontrado el port base de la GUS y puede abandonar la rutina de detección. De lo contrario ha de repetirlo todo para el siguiente port posible. No olvide terminar la rutina igualmente después del test del port *280h*. En ese caso no hay ninguna GUS instalada en el ordenador y el test de otros puertos no le llevará tampoco a ninguna parte. Aquí tiene el código de cómo podría ser esta rutina:

```
detect_gus proc near
;  ##################################################################
;  ### La rutina sirve para reconocer la Gravis Ultrasound. Se    ###
;  ### reconoce el puerto base. La función devuelve 0 si se       ###
;  ### encuentra la tarjeta; si no, 1.                            ###
;  ##################################################################
  mov di,1F0h
@detect_loop:                    ; probar puertos posibles con bucle
  add di,10h
  mov dx,di
  add dx,103h                    ; probar inicialización
  mov al,4Ch
  out dx,al
  mov dx,di
  add dx,105h
  mov al,0
```

```
        call u_delay
        call u_delay
        mov dx,di
        add dx,103h
        mov al,4Ch
        out dx,al
        mov dx,di
        add dx,105h
        mov al,1
        mov dx,di                       ; intentar escribir datos a la
        add dx,103h                     ; GUS-RAM
        mov al,43h
        out dx,al
        mov dx,di
        add dx,105h
        mov al,0h
        out dx,al
        mov dx,di
        add dx,103h
        mov al,44h
        out dx,al
        mov dx,di
        add dx,105h
        mov al,0h
        out dx,al
        mov dx,di
        add dx,107h
        mov al,0AAh
        out dx,al
        call u_delay                    ; esperar, para que el GF1
        call u_delay                    ; no se pueda entrometer
        call u_delay
        call u_delay
        call u_delay
        call u_delay
        xor ax,ax                       ; leer valor de la RAM-GUS
        mov dx,di
        add dx,107h
        in al,dx
        cmp al,0AAh                     ; valor leído=valor escrito?
        je @Karte_encontrado            ; yupiii, tarjeta encontrada!
        cmp di,280h
        jae @Karte_nicht_encontrado     ; tarjeta no está en el puerto :(
        jmp @detect_loop                ; probar otro
@Karte_encontrado:
        mov w u_base,di                 ; inicializar reg. base de tarjeta
        mov ax,di
        add ax,6
```

```
  mov w u_status,ax
  mov ax,di
  add ax,102h
  mov w u_voice,ax
  mov ax,di
  add ax,103h
  mov w u_command,ax
  mov ax,di
  add ax,104h
  mov w u_Datalo,ax
  mov ax,di
  add ax,105h
  mov w u_Datahi,ax
  mov ax,di
  add ax,107h
  mov w u_DramIO,ax
  mov ax,0
  jmp @Fin_reconoc_tarjeta
@Karte_nicht_encontrado:
  mov ax,1
@Fin_reconoc_tarjeta:
  ret
detect_gus endp
```

El código de la subrutina de espera es el siguiente:

```
u_delay proc pascal
;  ################################################################
;  ### espera el tiempo necesario para double-writes          ###
;  ################################################################
  mov dx,300h
  in al,dx
  in al,dx
  in al,dx
  in al,dx
  in al,dx
  in al,dx
  in al,dx
  ret
u_delay endp
```

Cargar datos de sonido

La carga de datos en la RAM de la GUS es bastante sencilla. Los datos se copian en un bucle de la memoria principal en la DRAM de la Gravis Ultrasound. Para mantener rápida la rutina de carga, se debería escribir con el mínimo posible de comando *out* por pasada.

En Ultra_Mem2Gus puede encontrar un ejemplo de rutina de carga. A esta se le ha de pasar un puntero a la zona fuente en la RAM, la posición en la RAM de la GUS y la longitud del bloque de datos a copiar.

La rutina carga primero el puntero a la zona fuente, y determina la posición inicial *HI* en la RAM de la GUS. Esta posición no se modifica en cada ciclo, sino sólo cada 65535 veces. Por ello es importante que no vuelva a definir en cada ciclo de nuevo este valor. A continuación se carga la longitud y finalmente se recorre el bucle de copia en sí (Copy-Loop) *Longitud* veces. En el bucle ha de fijar primero la posición inicial *LO* de la DRAM de la GUS, cargar el byte de la zona fuente y escribirlo a continuación mediante la función *Poke* a la RAM de la GUS. Como la dirección *HI* no se fija en cada pasada, ha de comprobar si la posición *LO* ha alcanzado el valor *FFFFh*. En ese caso se realizará un rebase para el siguiente byte, y la posición *HI* se ha de fijar de nuevo. El aspecto que puede tener semejante rutina de copia en la práctica lo podrá ver a continuación:

```
Ultra_Mem2Gus proc pascal sampp:dword,start:dword,longitud:word
;   ##############################################################
;   ### copia zona de memoria de la RAM a la GUS RAM          ###
;   ##############################################################
    push ds
    push si
    mov si,[bp+12]                  ; Segmento
    mov ds,si
    mov si,[bp+10]                  ; Offset
    mov dx,w u_Command              ; fijar Hi-byte direcc. GUS-DRAM
    mov al,44h
    out dx,al
    mov dx,w u_DataHi
    mov ax,[bp+08]                  ; hstart
    out dx,al
    mov cx,[bp+4]                   ; cargar longitud
@Copy_loop:
    mov dx,w u_Command              ; fijar Lo-byte direcc. GUS-DRAM
    mov al,43h
    out dx,al
    mov dx,w u_DataLo
    mov ax,[bp+06]                  ; lstart
    out dx,ax
    mov dx,w u_DramIo               ; cargar byte
    lodsb
    out dx,al
    cmp word ptr [bp+06],0ffffh     ; lstart = 0ffffh ?
    je @rebasamiento
    inc word ptr [bp+06]            ; lstart++
    jmp @seguir
@rebasamiento:
```

```
    inc word ptr [bp+08]          ; hstart ++
    mov word ptr [bp+06],0        ; lstart a 0
    mov dx,w u_Command            ; fijar Hi-byte direcc. GUS-DRAM
    mov al,44h
    out dx,al
    mov dx,w u_DataHi
    mov ax,[bp+08]                ; hstart
    out dx,al
@seguir:
    loop @copy_loop
    pop si
    pop ds
    ret
Ultra_Mem2Gus endp
```

Reproducir Muestreos

Ahora que hemos cargado nuestro muestreo en la DRAM de la GUS, vamos a reproducirlo. Pero antes de poder iniciar la reproducción en un canal, se han de determinar los parámetros específicos de canal. Primero deberíamos elegir el volumen deseado para el canal. Esto se realiza con el comando de volumen del registro de comandos. En U_VoiceVolume puede encontrar un ejemplo del aspecto que podría tener el procedimiento para ajustar el volumen.

```
u_VoiceVolume proc pascal Nro:byte, Vol:word
; ******************************************************************
;   *** Fija el volumen para un canal (0 - 63)                   ***
; ******************************************************************
    mov dx, w u_Voice             ; Elegir voz
    mov al, Nro
    out dx, al
    mov dx, w u_Command           ; Comando fijar volumen
    mov al, 9
    out dx, al
    mov dx, w u_DataLo            ; Cargar volumen GUS de la tabla
    mov di, vol                   ; y fijarlo
    shl di, 1
    mov ax, word ptr [offset uVolumes + di]
    out dx, ax
    ret
u_VoiceVolume  endp
```

Como siguiente paso se han de ajustar los parámetros generales para la voz correspondiente. Esto significa el inicio de la voz en la DRAM de la Gravis Ultrasound, la posición inicial del Loop y la posición final de la voz. El inicio de voz se determina mediante la combinación de comandos 0ah/0bh del registro de comandos. Si no quiere repetir la voz (looping) lo mejor el fijar el inicio del looping en el comien-

zo de la voz. De lo contrario escriba aquí la posición en la RAM de la GUS, y no un offset, calculada desde el inicio de la voz. La combinación para fijar el Loop-Start-Register es 02h/03h y para determinar la posición final necesita 04h/05h. El procedimiento U_Voicedata ofrece un ejemplo para la inicialización correcta de los registros.

```
u_Voicedata proc pascal start,lsta,llongitud:dword,Nr:word
;   ##################################################################
;   ### fija los parámetros para un canal                          ###
;   ##################################################################
    mov dx,w u_Voice                ; seleccionar voz
    mov ax,Nr
    out dx,al
    mov dx,w u_command              ; fijar inicio de voz
    mov al,0ah
    out dx,al
    mov ax, word ptr [start+2]
    mov cx, word ptr [start]
    mov bx,cx
    shr ax,7
    shr cx,7
    shl bx,9
    or  ax,bx
    mov dx,w u_DataLo
    out dx,ax
    mov dx,w u_Command
    mov al,0bh
    out dx,al
    mov dx,w u_datalo
    mov ax,word ptr [start]
    shl ax,9
    out dx,ax
    mov dx,w u_command              ; fijar inicio de bucle
    mov al,2
    out dx,al
    mov ax, word ptr [lsta]
    mov cx, word ptr [lsta+2]
    mov bx,cx
    shr ax,7
    shr cx,7
    shl bx,9
    or  ax,bx
    mov dx,w u_DataLo
    out dx,ax
    mov dx,w u_Command
    mov al,3
    out dx,al
    mov dx,w u_datalo
    mov ax,word ptr [lsta]
    shl ax,9
```

```
      out dx,ax
      mov dx,w u_command          ; fijar fin de bucle
      mov al,4
      out dx,al
      mov ax, word ptr [llongitud]
      mov cx, word ptr [llongitud+2]
      mov bx,cx
      shr ax,7
      shr cx,7
      shl bx,9
      or  ax,bx
      mov dx,w u_DataLo
      out dx,ax
      mov dx,w u_Command
      mov al,5
      out dx,al
      mov dx,w u_datalo
      mov ax,word ptr [llongitud]
      shl ax,9
      out dx,ax
    ret
  u_Voicedata endp
```

Ahora la GUS ya sabe en qué volumen ha de reproducir la voz y dónde se encuentran los datos de muestreo en la RAM de la Gravis Ultrasound. Pero le falta la información más importante: la frecuencia con la que se ha de reproducir la voz. Esta se puede ajustar mediante el comando *1* del registro de comandos. Sin embargo ha de tener en cuenta que no se pasa la frecuencia de salida, sino que su valor se calcula según la fórmula

```
Frec := Frecuencia DIV Voice_Divisor[num_voices].
```

El porqué de ello seguramente sólo lo saben Forte y Gravis... Un ejemplo de cómo puede ajustar correctamente una voz se encuentra en el procedimiento *U_Voicefreq*. El procedimiento ajusta primero el número de la voz a modificar. A continuación le indica al GF1 que se ha de modificar la frecuencia de la voz determinada. Después se determina la posición en la tabla de los Voice-Divisors. Esta posición depende del número de canales empleados. Como último paso el procedimiento carga el divisor correspondiente, divide la frecuencia en consecuencia y envía el resultado.

```
u_Voicefreq proc pascal Nro:byte, Freq:word
; ***************************************************************
; *** Ajusta la frecuencia con la que se reproduce el canal    ***
; ***************************************************************
  mov dx, w u_Voice                ; Direccionar voz
  mov al, Nro
```

```
    out dx, al
    mov dx, w u_Command                ; Comando escribir frec. voz
    mov al, 1
    out dx, al                         ; Frec := Frecuencia DIV
    xor bx, bx                         ;
  Voice_Divisor[num_voices-13]
    mov bl, num_voices
    mov ax, Freq
    mov di, bx
    sub di, 14
    xor bx, bx
    xor dx, dx
    mov bl, byte ptr [voice_Divisor+di]
    div bx
    mov dx, w u_DataLo
    out dx, ax
    ret
  u_Voicefreq endp
```

Una opción que no ha de emplear, pero no parece interesante mencionar, es la posibilidad de ajustar individualmente el balance para cada voz. Esto se puede realizar con el comando *0ch* del registro de comandos. Los valores válidos están entre *0* y *15*, donde el *0* representa la izquierda y *15* la derecha.

```
  u_VoiceBalance proc pascal Nro, balance : byte
  ; ******************************************************************
  ; *** Ajusta la Pan-Position para un canal (0 - 15)              ***
  ; ******************************************************************
    mov dx, w u_Voice                  ; Elegir voz
    mov al, byte ptr Nro
    out dx, al
    mov dx, w u_Command                ; Comando Set Pan-Position
    mov al, 0Ch
    out dx, al
    mov dx, w u_dataHi                 ; Escribir posición
    mov al, balance
    out dx, al
    ret
  u_VoiceBalance endp
```

Como último paso ha de iniciar la reproducción de la voz. Esto se regula mediante el registro de control de voces del registro de comandos. El significado de los diferentes bits lo puede encontrar en el apartado sobre el registro de control de voces. El procedimiento *U_StartVoice* se encarga de iniciar la voz. Tenga en cuenta que la reproducción comienza inmediatamente que haya modificado el registro. Si quiere detener la voz a mano, ha de colocar el bit 1 de este registro a *1*.

```
U_StartVoice proc pascal Nro, Modo : byte
;  ****************************************************************
;  *** Inicia la reproducción en un canal GUS                    ***
;  ****************************************************************
  mov dx, w u_voice                ; Elegir voz
  mov al, byte ptr Nro
  out dx, al
  mov dx, w u_command
  mov al, 0                        ; Voice Mode
  out dx, al
  mov dx, w u_DataHi
  mov al, Modo                     ; Fijar byte de modo
  out dx, al
  ret
U_StartVoice endp
```

Modificar voces - efectos de ramping

Para obtener una reproducción limpia del sonido, no debería colocar el volumen de la voz por encima de U_VoiceVolume, sino con el procedimiento Voice_Ramping que aquí presentamos. Este procedimiento no coloca la voz directamente al volumen de destino, sino que comienza con volumen 0 y va incrementando el volumen hasta alcanzar el de destino. Esto tiene la ventaja de que se suprime el posible chasquido que a veces se puede escuchar al inicio de una voz debido al súbito cambio de frecuencias.

Para obtener esto, el procedimiento primero selecciona la voz a modificar. A continuación fija el factor de Ramping mediante el comando *Volume-Ramp-Rate* del registro de comandos a la rampa más rápida posible. Ahora se adapta el volumen actual, es decir, se coloca al valor inicial *0*. También el volumen inicial del Ramping se pone a *0*. El volumen de destino se determina como en el procedimiento *u_VoiceVolume* y se inicializa mediante el comando *Volume-Ramp-End* del registro de comandos. Como último paso importante se ajusta la dirección del Ramping en el registro de control de volumen. Con ello comienza la rampa.

```
voice_rampin proc pascal Stimme:byte,vol : word;
; ##################################################################
; ### Alternativa para la fijación directa del volumen de una voz.###
; ### El player pierde algo de agresividad, pero los posibles     ###
; ### chasquidos se reducen sensiblemente.                        ###
; ##################################################################
  mov dx,w u_voice                 ; seleccionar voz
  mov al,byte ptr Stimme
  out dx,al
  mov dx,w u_command               ; fijar factor de ramping
  mov al,6
  out dx,al
```

```
  mov dx,w u_datahi
  mov al,00111111b
  out dx,al
  mov dx,w u_Command              ; adaptar volumen actual
  mov al,9
  out dx,al
  mov dx,w u_datahi
  mov al,00010010b
  out dx,al
  mov dx,w u_command              ; fijar volumen de ramping inicial
  mov al,7
  out dx,al
  mov dx,w u_datahi
  mov al,00010010b
  out dx,al
  mov dx,w u_command              ; fijar volumen de ramping final
  mov al,8
  out dx,al
  mov dx,w u_datahi
  mov di,word ptr vol
  shl di,1
  mov ax,word ptr [offset uVolumes + di]
  shr ax,8
  out dx,al
  mov dx,w u_command              ; direcc. ramping en Volume Control
  mov al,0dh                      ; fijar registros
  out dx,al
  mov dx,w u_datahi
  mov al,0
  out dx,al
  ret
voice_rampin endp
```

15. Soporte de sonido para sus programas

Todas las demos programadas por los Coders de este mundillo ofrecen hoy en día soporte de sonido, igual que todos los juegos de ordenador. Una buena salida de sonido es un elemento importante de un buen software. Por ello hablaremos de todos los aspectos importantes de la programación de sonidos en este capítulo, así como de los diferentes formatos de los mismos.

Para las dos tarjetas de sonido más empleadas, la tarjeta Sound Blaster y la tarjeta de sonido de los Freaks Gravis Ultrasound, encontrará un MOD-Player para probar en el CD del libro.

15.1 El formato de archivo MOD

El formato de archivo MOD, originalmente del Commodore AMIGA, encontró su camino hacia el PC gracias a la expansión de la tarjeta Sound Blaster. Con él se puede realizar música digital sin grandes requerimientos de memoria.

Esto se consigue sampleando sonidos "naturales" y empleando estos samples como instrumentos. Esto naturalmente suena mucho más real que la síntesis FM de la Sound Blaster. Poco a poco la "Sampled Wave Technology" va encontrando soporte de hardware, por ejemplo en la Wave Blaster de Creative Labs o la nueva Gravis Ultrasound. Pero nosotros nos habremos de conformar con el único canal de sampling de la Sound Blaster.

El formato MOD

Existen diferentes versiones del formato MOD, ya que justamente al principio cada programador de editores "cocinaba su propia sopa". El formato MOD descrito a continuación se ha de considerar como el formato estándar. Un archivo MOD estándar se puede dividir en tres partes:

1. La cabecera MOD

La cabecera mide 1084 bytes y contiene todas las informaciones importantes de la pieza.

BYTES 0 - 19 - Song Name

En los primeros 20 bytes del tema se encuentra en nombre del mismo como cadena C, el último byte ha de ser *0*.

BYTES 20 - 41 - Nombres del primer instrumento

En estos 22 bytes se encuentra el nombre del primer instrumento como cadena C en formato ASCII, el último carácter ha de ser un *0*.

BYTES 42 - 43 - Longitud del instrumento

En estos dos bytes se encuentra la longitud del primer instrumento. La indicación se refiere a palabras. Para obtener el número de bytes, el valor se ha de multiplicar por dos. Además se ha de tener en cuenta que aquí se trata de un archivo que originalmente procedía del Amiga. El procesador 68000 guarda un valor en la secuencia High-Byte Low-Byte, y no como los procesadores de la serie 80x86 en Low-Byte High-Byte. Por ello se han de intercambiar el Low-Byte y el High-Byte para obtener el valor.

BYTE 44 - Fine Tune

Sólo se necesitan los 4 bits inferiores. Indican un valor entre *-8* y *8*, que permite realizar el afinado del instrumento. Esta opción, sin embargo, se emplea muy poco.

BYTE 45 - Volume

Este byte indica el volumen por defecto del instrumento. Los valores permitidos están entre *0* y *64*.

BYTES 46 - 47 - Loop-Start

El formato MOD ofrece la posibilidad de repetir un sample. Los dos bytes designan el inicio del bucle. También aquí se ha de tener en cuenta que se trata de una indicación de palabra de Amiga, es decir, que se han de intercambiar el Low-Byte y el High-Byte y multiplicar el resultado por dos.

BYTES 48 - 49 - Longitud del bucle

En estos dos bytes se encuentra la longitud del bucle. También aquí se trata de una palabra de Amiga. Las informaciones de los bytes 20 hasta 49 repiten para los demás treinta instrumentos del formato nuevo MOD, o los 14 del formato antiguo. Aquí presupondremos el ya muy extendido nuevo formato MOD.

BYTE 950 - Longitud del tema

Este byte designa la longitud del tema en patrones. Define la longitud real del tema, y no el número de patrones definidos.

BYTE 951 - CIAA-Speed

Este byte es específico del Amiga y no tiene significado en el PC.

BYTES 952 - 1079 - Arreglos del tema

Cada uno de los 128 bytes puede adoptar un valor entre *0* y *63*, que designa el patrón a reproducir. De estos 128 bytes y la longitud del tema se obtienen los arreglos del tema.

BYTES 1080 - 1083 - Identificación

En estos cuatro bytes se encuentra la identificación del archivo MOD. Si se encuentra el valor *'M.K.'* o *'FLT4'* se trata de un archivo MOD de cuatro voces con 31 instrumentos. En el PC también se han generado nuevos formatos, así por ejemplo *'6CHN'* representa un archivo MOD de seis voces, y *'8CHN'* uno de ocho voces.

2. Los patrones MOD

BYTES 1084 - 2107

En este byte se encuentra definido el primer patrón. Una nota de un patrón se compone de 4 bytes, y hay definidas cuatro notas por línea. Cada patrón contiene 64 líneas. De ello resulta la longitud total de un patrón de 1024 bytes. Es algo difícil determinar la cantidad de patrones, se ha de calcular. Para ello se ha de restar la longitud de la cabecera de la longitud total del archivo MOD. A continuación se ha de sumar la longitud de los diferentes samples, y restarla del tamaño del archivo. El resto se ha de dividir por el tamaño del patrón (1024) y así se obtienen el número de los samples. Los patrones se pueden leer ahora sin ningún problema.

3. Los datos de sampling MOD

BYTES n1 - nx

A continuación de los patrones se encuentran los datos de sampling en el *Signed Raw Data Format*. Es decir, que se trata de datos puros de sampling de 8 bits, que adoptan valores entre *-127* y *128*. En el PC estos datos se definen sin embargo entre *0* y *255*. Por ello los datos de sampling se han de convertir antes de la salida.

El formato de una nota MOD

El formato de una nota MOD es bastante complejo, pero se puede entender bastante bien.

BYTE 1

Los 4 bits superiores (7 a 4) se han de interpretar como los High-Bits del número de instrumento. Los 4 bits inferiores (3-0) pertenecen a la especificación del tono.

BYTE 2

El byte 2 contiene, junto con los bits 3 a 0 del primer byte la especificación del tono, que se mide en samples por minuto. Las tasa de muestreo estándares son:

$036 H-4	$071 H-3	$0E2 H-2	$1C5 H-1	$386 H-0
$039 A#4	$078 A#3	$0F0 A#2	$1E0 A#1	$3C1 A#0
$03C A-4	$07F A-3	$0FE A-2	$1FC A-1	$3FA A-0
$040 G#4	$087 G#3	$10d G#2	$21A G#1	$436 G#0
$043 G-4	$08F G-3	$11d G-2	$23A G-1	$477 G-0
$047 F#4	$097 F#3	$12E F#2	$25C F#1	$4BB F#0
$04C F-4	$0A0 F-3	$140 F-2	$280 F-1	$503 F-0
$055 E-4	$0AA E-3	$153 E-2	$2A6 E-1	$54F E-0
$05A D#4	$0B4 D#3	$168 D#2	$2D0 D#1	$5A0 D#0
$05F D-4	$0BE D-3	$17d D-2	$2FA D-1	$5F5 D-0
$065 C#4	$0CA C#3	$194 C#2	$328 C#1	$650 C#0
$06B C-4	$0D6 C-3	$1AC C-2	$358 C-1	$6B0 C-0

BYTE 3

En los 4 bits superiores (7 a 4) encontramos los Low-Bits del número de instrumento. En los 4 bits inferiores se encuentra el efecto.

BYTE 4

El byte 4 contiene el operador del comando de efecto.

Efectos

Efecto 00 - Arpeggio (dos operandos)

El efecto de *Arpeggio* provoca que la nota no se toque constantemente en el mismo tono, sino en tres tonos diferentes. El primer parámetro designa la diferencia entre la primera pareja de tonos y el segundo parámetro la diferencia entre la segunda pareja de tonos, medida en semitonos.

Efecto 01 - Portamento up (un operando)

Mediante este efecto se aumenta constantemente el tono de una nota. La velocidad del incremento se determina por el operador.

Efecto 02 - Portamento down (un operando)

Mediante este efecto se disminuye constantemente el tono de una nota. La velocidad del decremento es determinada por el operador.

Efecto 03 - Portamento to Note (un operando)

A diferencia de los efectos 01 y 02, aquí se especifica la nota de destino. El operando determina de nuevo la velocidad.

Efecto 04 - Vibrato (dos operandos)

Con este efecto se puede obtener un Vibrato. El primer operando indica la velocidad y el segundo la profundidad del mismo.

Efecto 05 - Portamento to Note + Volume Sliding (dos operandos)

Con este efecto se puede realizar un *Portamento to Note* y disminuir simultáneamente el volumen. La parte superior del operando (*Xx*) indica la velocidad de portamento y la segunda parte (*xX*) la velocidad en la que se ha de disminuir el volumen.

Efecto 06 - Vibrato + Volume Sliding (dos operandos)

Con esto podrá realizar simultáneamente un vibrato y disminuir el volumen. El primer operando (*Xx*) indica la velocidad con la que ha de realizarse el vibrato y en el segundo operando (*xX*) se encuentra la velocidad con la que se ha de reducir el volumen.

Efecto 07 - Trémolo (dos operandos)

El efecto de trémolo es muy parecido al efecto de vibrato. Designa una vibración del volumen. Ya que de esta forma frecuentemente se obtiene un efecto de vibrato, las rutinas de reproducción para los efectos de vibrato y trémolo pueden ser iguales con frecuencia. Como primer parámetro (*Xx*) debe pasar la velocidad del trémolo y como segundo operando (*xX*) se espera la profundidad del trémolo.

Efecto 08 -no definido

Este es el único comando no definido, de modo que se puede emplear para rutinas propias.

Efecto 09 - Fijar Sample-Offset (un operando)

Este efecto sirve para iniciar la reproducción de un sample no desde el principio, sino desde la posición de offset pasada en adelante. El valor pasado en el operando designa los dos primeros números de la indicación de longitud del sample.

Efecto 10 - Volume sliding (un operando)

Con este efecto disminuye o aumenta el volumen de un instrumento. En el caso de un operador del tipo *n0* se incrementa el volumen con la velocidad *n*, en el caso de un operando del tipo *0n* se reduce el volumen con la velocidad *n*.

Efecto 11 - Position Jump (un operando)

Se interrumpe la reproducción del patrón actual y se continúa en el patrón especificado en el operando. El operando pasado designa la línea en la que se ha de continuar la reproducción.

Efecto 12 - Set Note Volume (un operando)

Con este efecto se puede seleccionar el volumen de una nota. El valor pasado en el operador puede estar entre *0* y *63*.

Efecto 13 - Pattern break (ningún operando)

Con este efecto provoca la interrupción de la reproducción del patrón actual. La reproducción se continúa en el siguiente patrón.

Efecto 14 - Diferentes comandos (dos operandos)

El efecto 14 es un efecto muy versátil. Tiene varias subfunciones. El primer operando es responsable de la selección del subefecto deseado, mientras que el segundo parámetro sirve para el subefecto.

Efecto 14.1 - Fineslide Up

Este efecto trabaja igual que el *Portamento up* normal. Sin embargo sólo se realiza una modificación del tono, al contrario que *Portamento*, donde el tono se modifica durante todo el tiempo de ejecución de la nota.

Efecto 14.2 - Fineslide Down

Este efecto funciona análogamente al efecto 14.1, con la diferencia de que se desliza hacia abajo en vez de hacia arriba.

Efecto 14.3 - Glissando Control

Glissando es efectivo en combinación con los comandos de *Portamento*. Si el *Glissando* está activado, no se realiza una modificación continua del tono, sino en pasos de semitono. Un *1* en el parámetro activa el *Glissando*, un *0* lo vuelve a desactivar.

Efecto 14.4 - Vibrato Forma de onda

Con este efecto puede elegir la forma de onda para el efecto de *Vibrato*. El valor por defecto 0 activa una onda senoidal, el parámetro *1* activa una disminución constante, 2 una onda cuadrada y *3* una onda aleatoria.

Efecto 14.5 - Fijar Finetune

Con este comando puede modificar el valor de ajuste fino. Emplee la siguiente tabla, para encontrar el valor deseado:

Finetune	Valor
+7	7
+6	6
+5	5
+4	4
+3	3
+2	2
+1	1
0	0
-1	15
-2	14
-3	13
-4	12
-5	11
-6	10
-7	09
-8	08

Efecto 14.6 - Pattern Loop

Con este efecto puede repetir una parte determinada de un patrón. El parámetro *0* denomina el inicio del bucle y cualquier otro parámetro (*n*) lleva a una repetición de n veces del rango desde el inicio del bucle hasta la posición actual, antes de continuar la reproducción.

Efecto 14.7 - Forma de onda Tremolo

Con este efecto puede elegir la forma de onda para el efecto de *Tremolo*. El valor por defecto *0* indica una forma de onda senoidal, el parámetro *1* por una disminución constante, 2 por una onda cuadrada y 3 por una onda aleatoria.

Efecto 14.10 - Fine Volumesliding up

Este efecto realiza un único *Volumesliding*. Funciona igual que el *Volumesliding* normal, pero no realiza cambios constantes, sino una sola vez.

Efecto 14.11 - Fine Volumesliding down

Este efecto trabaja de forma análoga al efecto 14.10, sólo que realiza un *Sliding down*, y no un *Sliding up*.

Efecto 14.12 - Acortar reproducción de nota

Este efecto provoca reproducciones de nota acortadas. El parámetro pasado designa el número de Ticks después de los cuales se interrumpirá la reproducción de la nota. *Ticks* es la velocidad determinada mediante *Set Speed*. Con *F06* se reproducirían seis Ticks. Si le pasa a este efecto el valor *4*, se reproducirían 2/3 de la nota.

Efecto 14.13 - Retardo de notas

Con este efecto puede producir un retardo durante la reproducción de la nota. El tiempo de retardo se pasa en el parámetro como Ticks. Este efecto se puede emplear muy bien para simulaciones de eco.

Efecto 14.14 - Retardo de patrón

Este efecto detiene la reproducción del patrón durante el tiempo especificado en el parámetro.

Efecto 15 - Set Speed (un operando)

Este efecto determina la velocidad con la que se reproduce el archivo MOD. Los valores permitidos para el operando están entre *0* y *31*.

El formato 669

Aparte del formato MOD estándar existen algunos otros formatos, que se habitualmente fueron establecidos por los grupos de vanguardia. El formato 669 proviene de Renaissance. Es muy parecido al estándar MOD, pero algo más compacto. También se puede dividir en tres partes:

La cabecera 669

La longitud de la cabecera 669 no es, a diferencia de la cabecera MOD, estática, sino variable. Se orienta en la cantidad de samples definidos.

Bytes 0 - 1 - Identificación

En estos dos bytes se encuentra la identificación del archivo 669. El valor de la palabra de identificación es *$6669*.

Bytes 2 - 110 - Song Message

En estos 108 bytes hay de líneas de 36 caracteres cada una como una especie de texto de información acerca del tema. Esta propiedad es única en el formato 669.

Byte 111 - Número de Samples

En este byte se encuentra el número de samples almacenados. Los valores permitido se encuentran entre *0* y *64*. El número máximo de samples es por ello justo el doble que en los archivos MOD estándar.

Byte 112 - Número de patrones

En este byte se encuentra el número de los patrones almacenados. Esto es cómodo, ya que no hay que calcular el número de patrones en este formato.

Byte 113 - Repeat-Start

Este byte designa con qué patrón comienza la repetición.

Bytes 114 - 242 - Arreglos del tema

En este array de 128 bytes se encuentran los arreglos del tema. La entrada correspondiente indexa el número de patrón a reproducir. De forma que se permiten valores entre *0* y *128*.

Bytes 243 - 371 - Lista de tempos

En este array de 128 bytes se encuentran las velocidades a las que se han de reproducir los diferentes patrones.

Bytes 372 - 500 - Lista de Break-Positions

En este array de 128 bytes se encuentran las Break-Positions de los diferentes patrones. La entrada de la lista designa la línea en la que se ha de terminar el procesamiento de los patrones. Los valores permitidos están entre *1* y *64*.

*Bytes 501 - 501+(Número de Samples * $19) - Informaciones de Sample*

En la siguiente sección puede encontrar las informaciones de sampling. La longitud de la sección depende del número de samples declarados. La longitud del campo declarado para cada sample es de *$19* bytes. Tiene la siguiente estructura:

```
13 Bytes Nombre de archivo del instrumento con cadena ASCII
1 dword (= longint) longitud del instrumento
1 dword Posición inicial del bucle de instrumento, desde el inicio
  del instrumento
1 dword Fin del bucle de instrumento, desde el inicio del instrumento
```

Los patrones 669

*Byte 501 + (Número de Samples * $19)*

Byte 501 - 501+(Número Samples$19)+(Número patrones*$600) - Patrones*

En la siguiente sección están definidos los siguientes patrones. Un patrón se compone de 64 líneas de 8 notas cada una. Cada nota mide 3 bytes. De ello resulta una longitud total de 1536 bytes por patrón. Las informaciones en estos 3 bytes están guardadas de forma que pueden "atravesar" los bytes (una parte de la información se encuentra en un byte, y el resto en el principio del siguiente). Los diferentes bits tienen el siguiente significado:

BYTE 0	76543210
BYTE 1	76543210
BYTE 2	76543210

Byte 0, Bit 7 hasta Byte 0, Bit 2 - Valor de nota

En estos 6 bits se encuentra almacenado el valor de nota de la nota a reproducir. Se obtiene según *VALORNOTA := (12 * Octava) + Nota*. La nota *D1* tendría, por tanto, el valor *12 * 1 + 2 = 14*. De esta forma se pueden direccionar cinco octavas.

Byte 0, Bit 1 hasta Byte 1, Bit 4 - Número de instrumento

En estos 6 bits se encuentra almacenado el número de instrumento a reproducir.

Byte 1, Bit 3 hasta Byte 1, Bit 0 - Volumen

En los cuatro bits se encuentra almacenado el volumen de la nota a reproducir. Sólo son posibles 16 volúmenes, pero esto es suficiente para el uso normal, ya que una diferenciación más fina sería difícil de distinguir.

Byte 2, Bit 7 hasta Byte 2, Bit 4 - Comando

En estos 4 bits se almacena el comando a aplicar. Los valores que se encuentren se han de interpretar de la siguiente forma:

Valor	Descripción
0	Portamento up
1	Portamento down
2	Portamento to Note
3	Pattern Break
4	Vibrato
5	Set Speed

Se ha de tener en cuenta que el operando de comando del vibrato se ha de interpretar según la fórmula *VALORVIBRATO := Operando SHL 4 + 1.*

Byte 2, Bit 3 hasta Byte 2, Bit 0 Operando de comando
En los últimos 4 bits está almacenado el valor para el operando del efecto.

Valores especiales
Las siguientes combinaciones de bytes tienen un significado especial:

Byte 0 = $FE	Sin nota, sólo cambio de volumen
Byte 0 = $FF	Sin nota y sin cambio de volumen
Byte 2 = $FF	Sin comando

El formato de archivo Scream Tracker

Otro formato bastante popular es el formato del Scream Trackers de Future Crew. El problema de este formato es que las informaciones disponibles sobre él son muy difíciles de obtener. En el Scream Tracker se distingue entre temas y módulos. La diferencia estriba en si se almacenan instrumentos (módulos) o no. Pero como sólo los módulos han tenido una verdadera difusión, se tratarán aquí con preferencia.

La cabecera STM

La cabecera de los archivos STM que se pueden ser creados con en Scream Tracker V2.24, tiene la siguiente estructura:

Bytes 0 - 19 - Song-Name
En los primeros 20 bytes se guarda el nombre del tema en formato ASCIIZ. Resuma el nombre para procesarlo como *Array of char*.

Bytes 20 - 27 - Trackername

En estos 8 bytes se encuentra el nombre del tracker con el que se creó el tema. La cadena no está terminada en *0*.

Byte 28 - Versión

Aquí se puede encontrar el número de versión del archivo. Actualmente es *$1A*.

Byte 29 - Tipo de archivo

En este byte se indica de qué tipo de archivo se trata. *1* indica un tema (sin samples), 2 por un módulo (con samples).

Bytes 30 - 32 - Número de versión

En estos dos bytes se encuentra en número de versión del archivo. En el byte 30 se almacena el número principal y en el byte 31 el número secundario de versión.

Byte 32 - Tempo

Aquí puede encontrar la indicación de en qué tempo se ha de reproducir el archivo.

Byte 33 - Número de patrones

En este byte está el número de patrones almacenados. Es muy importante, ya que se necesitará después al cargar el patrón. Es muy cómodo, por que no se ha de calcular el número de patrones como en los archivos MOD.

Byte 34 - Volumen global

Aquí se encuentra el volumen en el que se ha de reproducir el archivo. Los valores permitidos están entre *0* y *63*.

Bytes 36 - 47 - Reserved

Los bytes 36 hasta 47 están reservados para propósitos del Scream Tracker.

Bytes 48 - 80 - Instrumento Nº 1

En los siguientes 32 bytes están almacenados los datos del primer instrumento. Las informaciones de instrumento están repartidas de la siguiente forma:

Bytes 48 - 59 - Nombre de instrumento

En los bytes 48 hasta 59 está almacenado el nombre del primer instrumento como texto ASCII.

Byte 60 - Número de instrumento Zero

El byte 60 siempre es *0* y termina el nombre del instrumento (ASCIIZ).

Byte 61 - Unidad de instrumentos

En este byte se encuentra indicado el número de la unidad en la que se encuentra almacenado el instrumento. Es importante para los temas (sin samples).

Bytes 62 - 63 - Reservados

Estos dos bytes están reservados y se emplean internamente como dirección de segmento.

Bytes 64 - 65 - Longitud del instrumento

En estos dos bytes puede encontrar la longitud del instrumento. La indicación se refiere a bytes.

Bytes 66 - 67 - Loop Start

En los bytes 66 y 67 está almacenada la posición inicial del bucle en los datos de sampling, en la que se puede reproducir el instrumento.

Bytes 68 - 69 - Loop End

Estos dos bytes contienen la posición final del bucle en los datos de sampling.

Bytes 70 - Volumen

Aquí se encuentra el volumen por defecto del instrumento. Puede adoptar valores entre *0* y *63* y ser sobrescrito en tiempo de ejecución.

Byte 71 - Reservado

Este byte está reservado.

Bytes 72 - 73 - Velocidad para C1

En estos dos bytes se debería encontrar la velocidad para C1. Realmente el valor de estos bytes suele ser *0*.

Bytes 74 - 77 - Reservados

También estos cuatro bytes están reservado para propósitos internos.

Bytes 78 - 79 - Dirección de segmento / Longitud en párrafos

Aquí puede la dirección de segmento interna en un tema. En los módulos (más comunes) se encuentra en este lugar la longitud en párrafos (longitud en bytes

DIV 16). Estas informaciones se repiten para los demás 30 instrumentos restantes. Así que continuamos con

Bytes 1028 - 1091 Arrangement

En este array de 64 bytes se puede encontrar el arreglo de la pieza. Cualquier entrada de esta zona indica el número del patrón correspondiente.

Bytes 1092 - 2115 Patrón 1

En los bytes 1092 hasta 2115 se encuentra almacenado el primer patrón. Se compone de 64 líneas de 4 notas cada una. Una nota está formada por 4 bytes. De esto resulta un tamaño de patrón de 1024 bytes (64 * 4 * 4). Las notas tienen la siguiente estructura:

Byte 0, Bits 7 hasta 4 - Octava

En estos 4 bits puede encontrar el número de la octava de la nota a reproducir.

Byte 0, Bit 3 hasta 0 - Nota

Aquí se encuentra la nota a reproducir. Con esta relación *0 = C, 1 = C#, 2 = D* etc.

Byte 1, Bit 7 hasta 3 - Instrumento

En estos 5 bits se encuentra almacenado el número del instrumento a reproducir.

Byte 1, Bit 2 hasta Byte 2, Bit 4 - Volumen

En estos bits se puede encontrar el volumen del instrumento en ejecución.

Byte 2, Bits 3 hasta 0 - Efecto

El número del efecto actual se encuentra en estos 4 bits. Los números corresponden a los efectos presentados en los archivos MOD.

Byte 3 - Operandos

En este byte se almacenan los operandos del efecto. Después de este patrón siguen los demás patrones definidos. Su número se encuentra en el número de patrones en la cabecera. Si se tratara de un módulo, tras los patrones se encuentran los datos de sampling. Su longitud se encuentra guardada en la cabecera.

El formato S3M

El formato S3M es el formato de sonido empleado actualmente por la Future Crew. Procede del Scream Tracker 3.0 y probablemente es uno de los formatos de sonido

más flexibles. Puede guardar tanto módulos de sampling como piezas de Adlib. Diferencia entre módulos que contienen samples y temas que sólo guardan los arreglos.

La cabecera S3M

Bytes $00-$1B - Song-Name

Al principio de la cabecera se encuentra el nombre de la pieza. El nombre se termina en *0*.

Byte $1C - Identificación $1A

Este byte siempre contiene el valor *$1A*. Simplemente sirve para la identificación y no tiene otras funciones importantes.

Byte $1D - Tipo de archivo

El tipo de archivo decide si hay presente un archivo de módulo, o uno de tema. El valor *16* indica archivo de módulo y el valor *17* por un archivo de tema.

Bytes $1E - $1F - No empleado

Bytes $20 - $21 - Longitud del arreglo

En esta variable de palabra se encuentra la longitud variable del arreglo. Esta longitud siempre ha de ser un número par.

Bytes $22 - $23 - Número de instrumentos

En esta palabra se guarda el número de los instrumentos empleados en la pieza.

Bytes $24 - $25 - Número de patrones

Aquí se encuentra el número de patrones definidos. El valor se necesita además para la obtención de la longitud del Parapointer de patrones.

Bytes $26 - $27 - Flags

Las banderas (flags) influencian la inicialización de diferentes efectos. Los bits tienen el siguiente significado:

Bit 1	ST2-Vibrato
Bit 2	ST2-Tempo
Bit 3	Emplear Amiga-Sliding
Bit 4	Optimizaciones para volumen 0
Bit 5	Mantener límites de Amiga
Bit 6	Permitir efectos de filtro y sonido

Bytes $28 - $29 - Informaciones de versión

En estos dos bytes se almacena la información de versión del archivo. El número del tracker se encuentra en los 4 bits superiores, el número de versión está en los 12 bits inferiores. El valor para el Scream Tracker 3.0 es por ello *$1300*.

Bytes $2A - $2B - Versión del formato de archivo

Actualmente hay dos versiones distintas del formato de archivo. Un valor *1* señaliza que se trata de un archivo original. En el caso del valor 2 también es un archivo original, pero los samples se encuentran archivados en formato *unsigned*, tal y como lo conocemos del PC.

Bytes $2C - $2F - Identificación

Los bytes contienen la identificación 'SCRM', con la que se puede identificar el archivo Scream Tracker.

Bytes $30 - Volumen principal

En este byte se guarda el volumen principal empleado durante el inicio.

Byte $31 - Velocidad inicial

La velocidad empleada inicialmente se encuentra en este byte (comando *A*).

Byte $32 - Tempo inicial

En este byte se guarda el tempo empleado durante el inicio (comando *T*).

Byte $33 - Master Multiplier

El valor del Master Multiplier se guarda en los 4 bits inferiores. Puede obtenerlo mediante un *AND* con *$0F*. Si los 4 bits superiores son distintos de 0, se empleó estéreo.

Bytes $34 - $3F - No empleado

Bytes $40 - $5F - Ajustes de canal

En estos 32 bytes se encuentran los ajustes para los 32 posibles canales. Cada byte correspondiente tiene el siguiente significado:

Valor	Significado
0	Sample en el canal izquierdo (S1)
1	Sample en el canal izquierdo (S2)
2	Sample en el canal izquierdo (S3)
3	Sample en el canal izquierdo (S4)
4	Sample en el canal derecho (S5)
5	Sample en el canal derecho (S6)
6	Sample en el canal derecho (S7)
7	Sample en el canal derecho (S8)
8	Voz de melodía AdLib, izquierda (A1)
9	Voz de melodía AdLib, izquierda (A2)
10	Voz de melodía AdLib, izquierda (A3)
11	Voz de melodía AdLib, izquierda (A4)
12	Voz de melodía AdLib, izquierda (A5)
13	Voz de melodía AdLib, izquierda (A6)
14	Voz de melodía AdLib, izquierda (A7)
15	Voz de melodía AdLib, izquierda (A8)
16	Voz de melodía AdLib, izquierda (A9)
17	Voz de melodía AdLib, derecha (B1)
18	Voz de melodía AdLib, derecha (B2)
19	Voz de melodía AdLib, derecha (B3)
20	Voz de melodía AdLib, derecha (B4)
21	Voz de melodía AdLib, derecha (B5)
22	Voz de melodía AdLib, derecha (B6)
23	Voz de melodía AdLib, derecha (B7)
24	Voz de melodía AdLib, derecha (B8)
25	Voz de melodía AdLib, derecha (B9)
26	Adlib Basedrum, izquierda (AB)
27	Adlib Snare, izquierda (AS)
28	Adlib Tom, izquierda (AT)
29	Adlib Cymbal, izquierda (AC)
30	Adlib Hihat, izquierda (AH)
31	Adlib Basedrum, derecha (BB)
32	Adlib Snare, derecha (BS)
33	Adlib Tom, derecha (BT)
34	Adlib Cymbal, derecha (BC)
35	Adlib Hihat, derecha (BH)
>128	No activo
255	No empleado

Bytes $60 - ($60 + Longitud del arreglo) - Arreglo

Aquí se encuentra el arreglo de la pieza. Los valores guardados están por el número del patrón a reproducir.

Bytes xxxx - Parapointer de instrumentos

A continuación del arreglo se encuentran los Parapointer a los diferentes instrumentos. Bajo Parapointer (Paragraph-pointer) se entiende el Offset, calculado desde

el principio de la cabecera, DIVidido entre 16. Un Parapointer mide una palabra, de modo que la longitud total de los Parapointer a los instrumentos es de (*Número de instrumentos*) * 2.

Bytes xxxx - Parapointer de patrones

A continuación del Parapointer de instrumentos vienen los Parapointers de los diferentes patrones. Designan el principio de cada uno de los patrones en el archivo. La longitud de los Parapointer es (*Número de patrones*) * 2.

Los patrones S3M

La longitud de un patrón se obtiene del número de *canales empleados * 320 bytes por canal*. Las notas de las diferentes líneas están guardadas de forma secuencia. Cada nota se compone de 5 bytes. Los bytes tienen el siguiente significado:

Byte	Significado
Byte 0	Nota, Bit7-4 = Octava Bit4-0 = Nota 255 = vacío 254 = Key Off (para Adlib)
Byte 1	Instrumento Nº, 255 = vacío
Byte 2	Volumen, 255 = vacío
Byte 3	Comando, 255 = vacío
Byte 4	Parámetro de comando

Los instrumentos S3M

El formato S3M diferencia entre dos tipos de instrumento distintos. Por una parte emplea los instrumentos de sampling conocidos de los archivos MOD. Pero además también se soportan instrumentos Adlib. Ambos tipos de instrumento se componen de una cabecera de *$40* bytes, a la que siguen los datos de sampling en el caso de los instrumentos de sample. Consideremos primero la estructura de la cabecera de los instrumentos de sample:

Byte $0 - Tipo de instrumento

Si en este byte encuentra el valor *1*, se trata de un instrumento de Sample. De lo contrario es un instrumento AdLib.

Bytes $1 - $C - Nombre de archivo DOS

El nombre de archivo del archivo de sample está guardado en estos bytes. Sin embargo NO se trata de una cadena terminada en cero.

Bytes $D - $F - Segmento de memoria

En el segmento de memoria está depositada la posición de párrafo (*Offset DIV 16*) de los datos de sampling.

Bytes $10 - $13 - Longitud de sample

En esta doble palabra se guarda la longitud de los datos de sampling. Se recomienda comprobar si la longitud abandona la zona de palabra, para trabajar con variables de palabra en el caso necesario.

Bytes $14 - $17 - Inicio del bucle

En este DoubleWord se guarda la posición del inicio del bucle en los datos de sampling.

Bytes $18 - $1B - Fin del bucle

En esta doble palabra puede encontrar la posición final del Looping en los datos de sampling.

Bytes $1C - Volumen

El volumen por defecto del instrumento se guarda en este byte.

Byte $1D - Disco

En este byte se informa de en qué unidad se encuentra un instrumento.

Byte $1E - Tipo de compresión

Este byte especifica si los datos de sampling están comprimidos o no. *0* indica datos de 8 bits sin comprimir, *1* indica el algoritmo de compresión DP30ADPCM1.

Byte $1F - Flags

En las banderas se encuentran informaciones más extensas sobre el tipo de los datos de sampling. Si el bit 1 está a uno, el Looping se encuentra activo. El bit 2 indexa los datos estéreo y el bit 3 representa los samples de 16 bits en formato Intel.

Bytes $20 - $23 - C2 Speed

Este valor se emplea para afinar el instrumento. Designa la velocidad a la que se ha de reproducir el sample, para obtener el tono C2.

Bytes $24 - $27 - No empleado

Bytes $2B - $29 - Posición en la GRAVIS-RAM

Esta palabra sólo es necesitada por el Scream Tracker 3.0. Designa la posición del sample en la RAM de la Gravis Ultrasound *DIV 32*.

Bytes $2A - $2F - No conocidos exactamente

Bytes $30 - $4B - Nombre de instrumento

Aquí se almacena el nombre del instrumento como cadena ASCII terminada en cero con una longitud máxima de 28 caracteres.

Bytes $4C - $4F - Identificación

Aquí se encuentra la identificación 'SCRS' de *Scream Tracker Sample*. Hablemos ahora de los instrumentos Adlib. La estructura de la cabecera es fundamentalmente similar a la del instrumento de sampling:

Byte 0 - Tipo de instrumento

Aquí puede encontrar el tipo de instrumento. Se aplica lo siguiente:

Valor	Significado
2	Adlib Melody
3	Adlib Basedrum
4	Adlib Snare
5	Adlib Tom
6	Adlib Cymbal
7	Adlib Hihat

Bytes $1 - $C - Nombre de archivo DOS

Aquí se encuentra el nombre de archivo del instrumento Adlib.

$D - $F - Bytes vacíos

Los bytes *$D* hasta *$F* contienen el valor 0.

$10 - $1B - FM Synthese Table

Los bytes *$10* hasta *$19* contienen alternativamente el modulador y la portadora para la síntesis *FM*. Las informaciones sobre ello lamentablemente se han de cuestionar parcialmente. Los valores para los bytes siguen las siguientes fórmulas:

```
Bytes $10 & $11 = (Frequencemultiplier) + (sustain * 32) + (pitch
vibrato * 64) + (volume vibrato * 128)
```

```
Bytes $12 & $13 = (63 - volume) + ((levelscale AND 1) * 128) +
((levelscale AND 2) * 64)
Bytes $14 & $15 = (Attack * 16) + decay
Bytes $16 & $17 = ((15 - sustain) * 16) + release
Bytes $18 & $19 = wave select
Byte $1A = (modulation feedback * 2) + additive sythesis
Byte $1B = no empleado
Byte $1C - Volumen
```

El volumen por defecto del instrumento se guarda en este byte.

Byte $1D - Disco

Este byte contiene la información de la unidad de disco en la que se puede encontrar un instrumento.

Bytes $1E - $1F - No empleados

Bytes $20 - $23 - C2 Speed

Este valor se emplea para el afinado del instrumento. Designa la velocidad con la que se ha de reproducir el instrumento, para obtener el tono C2.

Bytes $24 - $2F - No empleados

Bytes $30 - $4B - Nombre de instrumento

Aquí se guarda el nombre del instrumento como cadena ASCII terminada en cero y de una longitud máxima de 28 caracteres.

Bytes $4C - $4F - Identificación

Aquí se encuentra la identificación 'SCRI' de *Scream Tracker Instrument*.

15.2 Un MOD-Player para la Sound-Blaster

En este libro ha aprendido cómo programar demos gráficas. ¿Pero qué sería de la mejor demo sin una banda sonora "cañera"? Así que se necesita un reproductor de sonido (Sound Player). La posibilidad más sencilla sería un VOC-Player. Pero con un gasto de 16.000 bytes por segundo, ya quemarían 2,3 MBytes para los datos de sonido de una demo de dos minutos y medio. Incluso con loopings y compresión de datos no se podría reducir este valor a una medida razonable (hasta 400 KBytes). La posibilidad VOC sólo se puede considerar para presentaciones comerciales o aplicaciones CD-ROM, en las que no importan un par de MBytes de más o de menos.

De modo que se necesita una alternativa que reúna una buena calidad con pocos requerimientos de memoria. Y así llegamos a los archivos MOD. Por desgracia, los reproductores para ellos siempre están muy ligados al hardware disponible. En este capítulo trabajaremos en un player para la tarjeta Sound Blaster y en el capítulo siguiente presentaremos uno para la GUS, mucho más adecuada para estos propósitos.

Pero vamos a por la versión Sound Blaster. La tarjeta Sound Blaster es posiblemente la tarjeta de sonido más extendida, gracias al buen trabajo publicitario de Creative Labs. La tarjeta es el antiguo estándar para demos y aunque su soporte siempre conlleva algunos problemas debería hacerlo en cualquier caso, siempre que se lo permitan el tiempo de procesamiento restante y la memoria libre.

"Realmente no debería ser ningún problema el reproducir MODs a través de la Sound Blaster. Al fin y al cabo, la reproducción de samples mediante DMA tampoco es *taaaan* complicada y tampoco necesita prácticamente de tiempo de cálculo." Lamentablemente esto es sólo parcialmente correcto. Veamos el bueno y viejo Amiga, del que procede el formato MOD. Para la salida de sonido disponía de cuatro DAC. Con estos cuatro canales realmente era sencillo reproducir los MOD, ya que todas las manipulaciones se podían limitar a modificaciones de frecuencia y volumen. Con la Sound Blaster la cosa es ligeramente diferente. Sólo tiene un DAC (en los modelos en estéreo son dos). Esto lleva a que ya no podemos reproducir los datos alegremente, sino que tenemos que ¡calcularlos todos completamente! Y eso, con una frecuencia de reproducción de 22 KHz al fin y al cabo son ¡22.000 datos por segundo!

Así que se necesita un player rápido, si queremos que nuestras historias sonoras funcionen en segundo plano, a la vez que los gráficos, sin grandes pérdidas de rendimiento. Esto nos lleva directamente a la siguiente limitación: Ya que, por razones de velocidad, hemos de emplear aritmética de coma fija, pero necesitamos una resolución de 24 bits, no se puede soportar un 286 con sus registros de sólo 16 bits. A pesar de que esto se puede realizar en un 286 con unos cuantos trucos, de todas formas éste ya es bastante flojo de por sí, y además pertenece a una especie en extinción. Así que las molestias y la pérdida de rendimiento que resultarían de su soporte no justificarían.

Si el player se quiere escribir en Turbo Pascal, el lenguaje en el que se programan la mayoría de las demos, se plantea la pregunta de cómo realizar de forma efectiva el manejo de los 32 bits. En las versiones de Pascal inferiores a la 8 lamentablemente aún no se soporta código de 32 bits, y una versión pura en Pascal sería horriblemente lenta. Así que sólo queda una cosa: la inclusión de un módulo de ensamblador externo. Aquí tenemos todas las posibilidades del mundo y el juego de instrucciones completo del 386 a nuestra disposición.

Principios fundamentales de un MOD-Player

Antes de dedicarnos a la programación de un MOD-Player, primero hemos de averiguar cómo funciona exactamente esta rutina. Y la primera pregunta debería ser: ¿Cómo realiza en realidad los diferentes tonos?

Los datos de sampling guardados en un archivo MOD representan las amplitudes de la oscilación que está almacenada. En el Amiga pueden tomar valores entre *-128* y *127*, en el PC están entre *0* y *255*. Es decir, que el valor en el que no se puede oír nada porque no tiene oscilaciones, en el Amiga se encuentra en *0* y en el PC en 127. De modo que para convertir un sample del formato de Amiga al formato de PC, se ha de incrementar el valor correspondiente en *127*. El hecho de que sólo se guarden las amplitudes nos simplifica las cosas. Haremos servir un efecto que seguramente conocerá de los buenos viejos tiempos de los discos de vinilo: si éstos se reproducen demasiado rápido, todo suena demasiado agudo, si se reproducen demasiado lentos, el soprano se convierte en bajo. Justamente esto es lo que hacemos con nuestros samples: los reproducimos con diferentes velocidades.

En la práctica no podemos modificar la frecuencia de sampling de la tarjeta Sound Blaster cada vez. Esto funcionaría como mucho para una voz, pero en el caso de 2 o cuatro voces tendríamos problemas. Por ello hemos de emplear un pequeño truco. La idea fundamental es la siguiente: Supongamos que reproducimos nuestro archivo MOD con una tasa de salida de 16 kHz. Esto significa que, cuando reproduzcamos un sample de 16 kHz, escucharemos su tono original. Ya que hay que reproducir exactamente 16.000 bytes, hemos de avanzar exactamente un 1 byte en el sample para determinar el byte a procesar. Si queremos reproducir el sample con 32 kHz, hemos de ejecutarlo con doble velocidad. Esto significa que en el tiempo del que disponemos ya no podemos reproducir todos los bytes, sino solamente cada segundo. Lo mismo ocurre con una reproducción de 8 kHz. Ahora tenemos tanto tiempo, que hemos de producir cada byte dos veces.

Es conveniente saber que la diferencia entre los diferentes semitonos siempre es la raíz doceava de dos. Esto significa para nosotros en la práctica, que hemos de reproducir 1,059463094 veces más rápido un tono que se encuentra un semitono por encima de su predecesor. En correspondencia, para un tono inferior se aplica la amplitud de paso 0,943874312. Y ahora justamente hemos llegado a lo que no necesitamos en nuestro programa. No sería nada ventajoso el calcular todas las multiplicaciones para los pasos en los samples, durante tiempo de ejecución. Es mejor emplear una tabla precalculada con los correspondientes pasos para los diferentes tonos. La diferencia entre los pasos siempre es la doceava raíz de 2.

El siguiente punto elemental son los volúmenes. Para ello debemos recordar que los valores de sampling son las amplitudes de la oscilación. El silencio es un amplitud de *0* (o bien *127*). Aquí ya podemos reconocer el sistema del volumen: según lo grande que sea la amplitud, así de fuerte suena nuestro sample. Si ahora queremos calcular los volúmenes de las diferentes voces en el archivo MOD, nos bastará una variable de palabra para los cálculos. Los volúmenes de un archivo MOD siempre se encuentran entre 0 y 63. De modo que simplemente hemos de multiplicar el byte del sample con el volumen, y dividirlo a continuación por el volumen máximo, *64*. En la práctica *64* es *2 elevado a 6*, con lo que la división se puede sustituir por *SHR 6*, lo que por cada proceso se ahorran 13 ciclos de reloj.

Mediante el volumen se puede realizar con sencillez un efecto como el vibrato. Simplemente se ha de aumentar o reducir el volumen en función de los valores leídos de la tabla para obtener la impresión de un efecto de vibrato. Naturalmente también puede realizar un vibrato semejante con la velocidad de sampleado. Si la aumenta y reduce periódicamente, también aquí obtendrá un efecto de vibrato. A usted le queda la elección de por qué opción se decide. El método del volumen necesita algo menos de tiempo de procesador (sólo trabaja con bytes) pero suena ligeramente peor, además de dar algunos problemas cuando se ha de combinar con efectos como el *Volume-Sliding*.

Mezclar datos de sonido - estructura de un procedimiento de mezcla

Después de haber conocido los fundamentos más básicos, vamos a dedicarnos a la realización del player. El papel central del player lo tiene el procedimiento de mezcla. Para un byte de salida ha de ser recorrida por cada voz activa. Si una voz pasa a ser inactiva, no se debería recorrer el procedimiento completo cada vez y cargar un byte vacío, sino simplemente saltarse el procesamiento de la voz, o llamar un procedimiento vacío.

Al inicio del procedimiento ha de comprobar si ha alcanzado el final de la voz. No debería comparar directamente con el final de la voz, sino con un valor poco antes del final de la misma. Los pocos bytes que le puedan faltar al sample no se notarán. Si compara con el final real, casi siempre obtendrá un *overflow* (rebase). Y este lleva a que mezcle datos aleatorios en sus datos de salida. Esta es una fuente para los *Ultraclics* (crujidos), que se pueden evitar. A continuación presentamos la estructura esquemática de un procedimiento de mezcla:

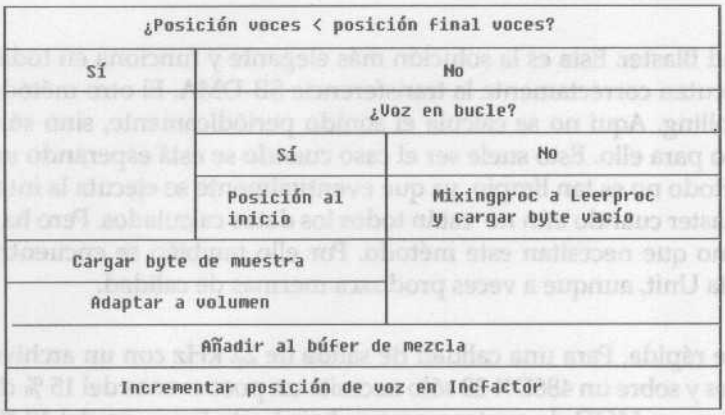

Figura 36: Estructura de un procedimiento de mezcla

Si no se ha alcanzado la posición final, puede proseguir normalmente con el procesamiento. De lo contrario ha de comprobar si la voz se repite (looping). Si es así, se ha de colocar la posición en la voz al valor inicial del bucle. De lo contrario el sample ha llegado a su fin. En este caso se carga un byte vacío (amplitud = *0*) y la variable de llamada del procedimiento de mezcla se desvía de la voz activa al procedimiento vacío anteriormente mencionado.

Después de haber realizado esta consulta de posición, ha de cargar los bytes del sampling del sample. Ha de adaptar el valor del byte al volumen de la voz, según el método anteriormente descrito. Ahora se puede sumar el byte al buffer de mezcla (*Mixingbuffer*). Este buffer se dividirá después por el número de voces, con lo que se obtiene una interpolación correcta de las voces.

Finalmente se ha se avanzar en la posición dentro del sample. Esto se obtiene mediante una sencilla suma del factor *Inc* con la posición actual. La posición y el factor *Inc* deberían ser valores de 32 bits, para poder calcular con una precisión de 16,16 bits.

MOD-Player para la tarjeta Sound-Blaster

El proyecto de desarrollar un MOD-Player está unido a un cierto gasto de tiempo. Especialmente los programadores no demasiado expertos se asustan a causa de ello, o incluyen muchos errores. A nosotros no pasó igual hace tiempo. Pero no tema, no habrá de volver a inventar la rueda dos veces. A lo largo de este capítulo conoceremos la Unit MOD_SB. Esta Unit le permite una sencilla inclusión en sus programas y es bastante flexible. Por una parte puede ejecutarla mediante la interrupción del temporizador. Éste es llamado periódicamente 50 veces por segundo y calcula una fracción de los datos a reproducir. La salida es controlada por la

interrupción Sound Blaster. Esta es la solución más elegante y funciona en todas las tarjetas que ejecutan correctamente la transferencia SB-DMA. El otro método disponible es el Polling. Aquí no se calcula el sonido periódicamente, sino sólo cuando hay tiempo para ello. Esto suele ser el caso cuando se está esperando un retrazado. Este método no es tan limpio, ya que eventualmente se ejecuta la interrupción Sound Blaster cuando aún no están todos los datos calculados. Pero hay partes en una demo que necesitan este método. Por ello también se encuentra implementada en la Unit, aunque a veces produzca mermas de calidad.

La Unit es bastante rápida. Para una calidad de salida de 22 kHz con un archivo MOD de ocho voces y sobre un 486DX-33 sólo necesita un poco menos del 15 % de tiempo de CPU. Con un MOD de cuatro voces se baja hasta la región del 13 %. Según lo intensivo de CPU que sea su programa puede emplear un archivo de cuatro u ocho voces. Si no ocupa todos los canales de un archivo de ocho voces, la salida será proporcionalmente más rápida. Como pequeño "Bonus" la Unit también contiene un rutina para reproducir archivos VOC. Con ello tendrá en esta Unit un conjunto de procedimientos universales para el control de sonido de la Sound Blaster.

El empleo de la Unit se explica en el programa MOD386. Es un pequeño player de MOD y VOC. Le puede pasar el nombre del archivo que ha de reproducir, o bien lo puede elegir en el menú que aparece. Si al iniciarlo especifica el parámetro de línea de comandos *-r*, se encontrará en el modo de repetición. Ahora podrá reproducir diferentes piezas musicales (MOD y VOC) una detrás de otra. Para averiguar cuánto tiempo de procesador sigue estando disponible, simplemente pulse la tecla [P] (de Performance) en el módulo. El player averigua entonces el tiempo disponible (esto tarda unos tres segundos) y le comunica los recursos remanentes del sistema. El rendimiento del sistema se comprueba averiguando cuánto tiempo le queda al sistema antes de un retrazado vertical. Si el sistema no hace nada más que esperar el retrazado e incrementar la variable de test, dispone del ciento por ciento del procesador. Este test se ejecuta cada vez al iniciar el programa. Si ahora se llama al test de rendimiento, el bucle en principio tampoco hace ninguna otra cosa que esperar el retrazado. La diferencia está en que en segundo plano se está reproduciendo el archivo MOD a través del temporizador. El cálculo de los datos necesita, según el ordenador empleado y la tasa de sampling ajustada, entre un diez y un treinta por ciento del rendimiento total del sistema. Tanto más bajo será el valor contado del bucle, del que se puede deducir directamente el rendimiento.

Variables necesarias del MOD-Player

Ahora que ya sabe lo que le espera en la Unit MOD_SB, vamos a explicar un poco más los detalles. Para poder sacarle todos el partido a la Unit, o incluso ampliarla (por ejemplo se podría añadir una pista de efectos de sonido...), ha de saber exac-

tamente lo que ocurre en la Unit, y qué se emplea para qué cosa. Por ello les presentaremos a continuación las variables principales con sus funciones, antes de entrar en la verdadera programación de la misma. Comencemos con la descripción de los tipos de que dispone la Unit:

VocHeader

En *VocHeader* se guardan las informaciones básicas de un archivo VOC. Al principio se encuentra la identificación de 20 bytes de longitud "Creative Voice File" + *1Ah*. A continuación se puede encontrar la posición a partir de la cual comienzan los datos de sampling, como palabra (Word). Después viene el número de versión del archivo VOC, primero el byte con el número alto y después el byte con el número bajo. La cabecera se cierra con un identificador encriptado del número de versión, que sin embargo no nos interesa en absoluto.

VoiceBlock

Los archivos VOC se dividen en diferentes bloques. Estos bloques comienzan con una cabecera que es leída en este tipo de datos. El primer byte contiene el identificador, los siguientes tres bytes la longitud. La tasa de muestreado está guardada de forma codificada en el siguiente byte. El último byte da informaciones sobre el tipo de compresión.

Pt

Pt es un tipo de datos simple, que simplifica un poco el manejo de los punteros. Contiene la palabra *ofs*, la parte de Offset del puntero y a continuación *sgm*, la parte de segmento del puntero. Y estos han sido todos los tipos que se emplean. De modo que pasemos a las constantes:

Incfacts : array[1..99] of longint;

Incfacts es una tabla de distancias de tono precalculadas. Los valores son de coma fija. Los 16 bits inferiores contienen la parte decimal del paso a dar y en los 16 bits superiores se guarda la mantisa del paso.

outfading : boolean = FALSE;

Si no quiere terminar su archivo MOD de forma abrupta, ha de colocar esta variable a *TRUE*. El volumen de salida se va reduciendo paulatinamente hasta que alcanza 0.

outvolume : byte = 63;

Outvolume designa el volumen con el que se reproduce el archivo MOD. El valor máximo está en *63*, el mínimo en *0* (= silencio). La variable se va decrementando si ha colocada *outfading* en *TRUE*.

MODDATSIZE : longint = 0;

Esta variable está pensada especialmente para el empleo de la Unit en demos o juegos. En ellos no se desea tener volando los archivos MOD por ahí como archivos individuales, sino que se empaquetan en un gran archivo de sonido. Pero naturalmente es necesario saber el tamaño de un archivo MOD. Por ello se ha de cargar esta variable con el tamaño del archivo MOD, si lo guarda en un gran archivo de sonidos. Si la variable tiene el valor *0*, se considera como tamaño del archivo MOD, la longitud del archivo en el que se encuentra éste.

Mastervolume : byte = 29;

Mastervolume es una variable que se emplea en tarjetas a partir de la Sound Blaster Pro. Estas tarjetas disponen de un chip mezclador que permite controlar el volumen por hardware. El volumen máximo es *31* y el mínimo *0*.

Startport : word = $200;

La tarjeta Sound Blaster se reconoce automáticamente. Para ello se comprueban los ports desde el port inicial al port final en pasos de *$10*. Si por ejemplo tiene instalada una tarjeta de red y no desea que se compruebe el port de la misma (esto podría llevar a algún conflicto) puede especificar los valores inicial y final en correspondencia. Lo mismo se aplica cuando quiere comprobar sólo *un* port, porque sabe que es el correcto. Simplemente iguale los valores inicial y final y sólo se comprobará un port.

Endport : word = $280;

Véase lo expuesto bajo *Startport*.

force_irq : boolean = FALSE;

Si coloca esta variable a *TRUE*, no se realiza ninguna detección de interrupción. En ese caso se emplea el valor por defecto 5.

force_base : boolean = FALSE;

Puede desconectar la detección del puerto base si coloca esta variable a *TRUE*. Se presupone el valor estándar *220h*.

interrupt_check : boolean = FALSE;

Esta constante global le indica al procedimiento de interrupción de la Sound Blaster, que se trata de un test de interrupciones y que no debe reproducirse ningún otro bloque.

timer_per_second : word = 50;

En la variable *timer_per_second* se guardan la cantidad de llamadas del temporizador por segundo.

In_retrace : boolean = FALSE;

Esta variable se pone a *TRUE* cuando el procedimiento *mod_waitretrace* está activo. Esto es necesario para apagar el procedimiento de mezcla si se está sincronizando con el retrazado vertical. Si no se desconectara, se llamará el procedimiento de mezcla durante la espera, lo que haría imposible la sincronización.

dsp_irq : byte = $5;

En la variable *dsp_irq* se guarda el número de IRQ de la Sound Blaster. Esta interrupción se ha de apuntar a una rutina propia de procesamiento. Si se apagó el reconocimiento de IRQ, puede indicar la interrupción asignándole a la variable el valor deseado.

```
dma_ch : byte = 1;
```

Aquí puede encontrar el número del canal DMA empleado. Si la tarjeta Sound Blaster utiliza otro canal que el *1*, ha de modificar esta variable.

```
dsp_adr : word = $220;
```

La variable *dsp_irq* contiene la dirección base de la tarjeta Sound Blaster. La variable se inicializa adecuada y automáticamente en el procedimiento de inicialización. Si desconecta la inicialización de la dirección base, ha de asignarle "a mano" el valor correcto a esta variable.

```
SbVersMin, SbVersMaj : BYTE = 0;
```

En estas dos variables la rutina de detección guarda el número de versión de la tarjeta Sound Blaster encontrada. En *SbVersMaj* puede encontrar el número de versión principal y en *SbVersMin* el número secundario de versión. La más importante de las dos variables seguramente es la versión principal, ya que la puede emplear, por ejemplo, para la identificación de la tarjeta.

```
SbRegDetected : Boolean = FALSE;
```

Esta variable es modificada por el procedimiento de inicialización. Si está a *TRUE*, se encontró una tarjeta Sound Blaster. De lo contrario puede interrumpir tranquilamente todas sus demás rutinas de Sound Blaster, porque no hay ninguna instalada.

```
SbProDetected : Boolean = FALSE;
```

También *SbProDetected* es modificado por el procedimiento de inicialización. Si se encontró una Sound Blaster Pro o una Sound Blaster 16, se coloca esta variable a *TRUE*.

```
Sb16Detected : Boolean = FALSE;
```

Si hay instalada una Sound Blaster 16 (ASP) en el ordenador, el procedimiento de inicialización coloca esta variable a *TRUE*. *SbProDetected* también está a *TRUE* en este caso.

MixerDetected : Boolean = FALSE;

Las tarjetas Sound Blaster a partir de la Sound Blaster Pro están equipadas con un chip mezclador. Si hay instalada una tarjeta a partir de la Sound Blaster Pro, se coloca esta variable a *TRUE*.

Voces : integer = 4;

En *Voces* se encuentra el número de las voces empleado en el archivo MOD. La variable puede tener el valor *4* o el valor *8* en esta versión de la Unit.

Modoktave : array[1..60] of word;

Los valores en este array corresponden a los tonos que están guardados en el archivo MOD. Así que si quiere determinar el valor de una nota, simplemente ha de determinar la posición del valor de la nota en la matriz (array).

dma_page : array[0..3] of byte;

Se soportan los canales DMA *0* hasta *3*. Esta variable se necesita para la programación directa de la transferencia DMA. Con ello se evita la inclusión de otra Unit.

dma_adr, dma_wc : array[0..3] of byte;

Véase *dma_page*.

sb16_outputlength : word = 0;

Si quiere reproducir datos en una Sound Blaster 16 mediante transferencia DMA, debería realizar la salida a partir del segundo bloque de datos mediante el comando DMA-Continue. Esto reduce los chasquidos DMA. Este método sin embargo sólo funciona con bloques de datos del mismo tamaño. En esta variable se guarda el tamaño del último bloque de datos reproducido. Si la longitud coincide con la del nuevo bloque a reproducir, se puede emplear el comando DMA-Continue.

ultima_salida : boolean = FALSE;

Coloque esta variable a *TRUE* para detener la salida mediante DMA. El procedimiento de interrupción no inicia la reproducción de un bloque nuevo cuando es direccionado.

Y estas han sido las constantes de la Unit. Finalmente llegamos a las variables. También aquí sólo hablaremos de las variables que tienen importancia. Las variables contadores o las que sirven como memorias temporales no se explicarán.

tamanyobloque : word;

El tamaño del bloque a reproducir se guarda en esta variable. Durante una interrupción siempre se calculan *tamanyobloque* datos.

dsp_rdy_sb16 : boolean;

Esta bandera se pone a *TRUE* cuando se termina la transmisión de datos vía DMA. Si se están transmitiendo datos, la variable contiene FALSE. No se moleste ni en este lugar ni en ningún otro de la coletilla *sb16*. Tiene su razón de ser en la historia de esta Unit. No significa que la variable sólo tenga significado para una Sound Blaster 16, sino que *también* sirve para una Sound Blaster 16.

OldInt : pointer;

En la variable *OldInt* se guarda la dirección de la antigua interrupción de temporizador, para poder restaurarlo al final del programa. Además también puede llamar la interrupción antigua mediante esta variable.

Irqmsq : Byte;

Esta variable es necesaria para el tratamiento de interrupciones. Con ella se des/enmascara la interrupción de la Sound Blaster.

Mezcla_proc, nmw_proc, dentro_proc : pointer;

Según si el archivo MOD tiene cuatro u ocho voces, se han de llamar distintos procedimientos. Para ahorrar tiempo y simplificar el acceso, no se ha de distinguir cada vez qué procedimiento se ha de llamar ahora. Sino que en la inicialización de

los punteros se asigna una sola vez la dirección de los procedimientos correspondientes. A través de estos punteros se pueden alcanzar estos procedimientos.

Activacion_nota : array[1..8] of integer;

Mediante esta variable se puede controlar un sencillo ecualizador. Los valores en el array están por el valor de cada canal. Se pone a 500 cada vez que se activa una nota de nuevo.

Rm : array[0..128] of Pointer;

La variable sirve para la gestión de los diferentes patrones. Los punteros apuntan al inicio del patrón correspondiente en la memoria.

Tema : array[1..128] of byte;

Este nombre de variable debería ser (al contrario que algunos otros) comprensible para todos, y no sólo para los que hemos desarrollado esta Unit. Aquí se encuentran los arreglos del tema. Los diferentes patrones se reproducen en la secuencia que aquí se determina.

Samp : array[1..64] of pointer;

Esta variable es un campo con punteros a los diferentes samples del archivo MOD en la memoria principal.

Sam_l : array[1..64] of word;

En este campo se encuentran las longitudes de los distintos samples.

loop_s, loop_l : array[1..64] of word;

En estos dos campos se guardan la posición inicial del looping (bucle) y la longitud del mismo.

In_St : array[1.8] of byte;

En este campo se encuentra el instrumento activo en cada momento con la voz correspondiente.

pat_anz : byte;

En la variable *pat_anz* se guardan el número de patrones del archivo MOD. No se trata de la longitud del tema, sino del número de patrones definidos.

Bucles_Sonido : byte;

Aquí se encuentra la cantidad de veces que se recorre el procedimiento de mezcla por llamada.

music_off : boolean;

Si esta variable se pone a *TRUE*, el procedimiento de interrupción no reproduce más música. Tampoco sigue calculando más música.

Notvol : array[1..8] of byte;

En la variable *Notvol* se encuentra el volumen del canal correspondiente. Se puede encontrar entre *0* y *64*.

Old_TContador : word;

Esta variable se necesita para la sincronización con la antigua interrupción de temporizador. Al fin y al cabo esta se ha de seguir llamando 18 veces por segundo. Esta variable se necesita para poder seguir la cuenta con la nueva interrupción del temporizador cuánto falta para la llamada de la antigua interrupción.

Song-Name : string[20];

En esta variable se guarda el nombre del archivo MOD.

NombresInst : array[1..31] of string[22];

En este campo se encuentran los nombres de los diferentes instrumentos. Frecuentemente se emplean estos nombres de instrumento para pequeños mensajes.

Inst_vol : array[1..31] of byte;

Aquí puede encontrar los volúmenes estándar del archivo MOD.

LongTema : byte;

En esta variable se guarda la longitud del tema en patrones. En los arreglos hay definidos una cantidad equivalente de entradas.

vocf : file;

Con esta Unit también puede reproducir archivos VOC. Puede acceder al archivo VOC mediante este Handle de archivo.

voch : vocheader;

Mediante *voch* puede acceder a la cabecera del archivo VOC. Su estructura la hemos conocido en el tratamiento de tipos.

vblock : voiceblock;

Mediante esta variable se direcciona el bloque activo en el archivo VOC.

SaveExitProc : pointer;

La Unit emplea un procedimiento de salida (ExitProc) propio. En él se reponen las variables modificadas y se liberan las zonas de memoria alojadas. Esta variable sirve para el almacenamiento de la dirección del antiguo *ExitProc*. ¡Al final hay que acordarse de llamarlo!

Portamento_Up_Voz : array[1..8] of longint;

En esta variable se almacena el valor de coma fija del posible *Portamento up*.

Portamento_Do_Voz : array[1..8] of longint;

En esta variable se almacena el valor de coma fija del posible *Portamento down*.

Mixingproc_Voz : array[1..8] of pointer;

Cada voz tiene su propio procedimiento de mezcla. Este se llama en dependencia del efecto activo en cada momento. El direccionamiento de este procedimiento mediante un puntero simplifica el acceso al mismo de forma considerable.

VozVacia : pointer;

Se trata de un puntero a un procedimiento que no hace nada. Si en una voz actualmente no se reproduce ningún sonido, se ejecuta este procedimiento como procedimiento de mezcla.

Efecto_Voz : array[1..8] of byte;

En cada voz puede haber activo un efecto determinado. El número de efecto activo actualmente se guarda en la variable correspondiente a la voz.

Longitud_Voz : array[1..8] of word;

La longitud de la voz se guarda en esta variable. Se necesita para la comparación de posición en el módulo de ensamblador.

Long_Bucle_Voz : array[1..8] of word;

La longitud del bucle de la voz se guarda en *Long_Bucle_Voz*. Los bucles sólo se tienen en cuenta si van más allá de 10 bytes.

Inicio_Bucle_Voz : array[1..8] of word;

Esta variable designa la posición en el sample en el que comienza el bucle. La posición final del bucle resulta de la suma de los valores del *Inicio_Bucle* y de la *Long_Bucle*.

Posicion_Voz : array[1..8] of longint;

La posición en una voz se guarda en esta variable como valor de coma fija. Los 16 bits superiores están por la posición en el muestreo, y los 16 bits inferiores designan los decimales.

Segmento_Voz : array[1..8] of word;

Al acceder al muestreo, se necesita el segmento del mismo. La variable Word se carga en el registro de segmento empleado para el acceso al muestreo.

Notvol_Voz : array[1..8] of word;

Esta variable se necesita en el módulo de ensamblador para el acceso al volumen de las diferentes voces.

Incval_Voz : array[1..8] of longint;

Cuánto se ha de incrementar la posición en el sample después de cada acceso, se encuentra en esta variable. También aquí se trata de un valor de coma fija. Y estas han sido las variables más importantes de la Unit. Esperamos que esta parte le sea útil como referencia y que ayude a la mejor comprensión de la Unit.

Control de tiempo del MOD-Player - Rutinas de temporizador

La reproducción del archivo MOD es un proceso periódico. En correspondencia se han de llamar regularmente las rutinas para mezclar la voz y para la reproducción. La rutina de salida se controla mediante la interrupción Sound Blaster. Sería posible dejar que la mezcla de las voces también pasara por esta interrupción, pero esto tardaría demasiado y llevaría a un desplazamiento irregular de los procesos que mientras tanto se desarrollan en la pantalla. Por ello se ha de dividir el procesamiento del bloque en varios sub-bloques más pequeños.

¿Pero cómo se puede garantizar que todos los bloques pequeños estén terminados de calcular cuando aparezca la interrupción Sound Blaster? Hemos de utilizar la interrupción del temporizador. Para empezar desviamos la interrupción a nuestra propia rutina. Ahora no nos basta con 18 llamadas por segundo. Necesitamos una frecuencia de entre 200 y 1000 interrupciones. Para ello hay que reprogramar el chip del temporizador. Esto se realiza calculando el valor que se le ha de pasar al chip mediante la siguiente fórmula:

```
Valor := 1193180 DIV Llamadas_por_segundo.
```

Como primer paso hemos de asignarle al Port *43h* el valor *3h*. Con ello sabe que se va a escribir un nuevo valor de temporización. A continuación enviamos primero el Low-Byte y después el High-Byte del valor calculado al Port *40h*. Con ello el temporizador ha quedado programado a la nueva frecuencia. Ahora sólo ha de desviar la rutina del temporizador a su propio controlador de interrupciones y ya puede comenzar. En esta Unit tiene sentido inicializar algunas variables necesitadas por el procedimiento de la interrupción. Esto evita algunos desagradables efectos secundarios. Veamos ahora la realización del procedimiento:

```
procedure AjustaTemporizador(Proc : pointer; Freq : word);
var icont : word;
    oldv : pointer;
begin;
 asm cli end;
 icont := 1193180 DIV Freq;
 Port[$43] := $36;
 Port[$40] := Lo(ICont);
 Port[$40] := Hi(ICont);
 Getintvec(8, OldV);
 setintvec(OldTimerInt, OldV);
 SetIntVec(8, Proc);
 old_tCont := 1;
 seccont   := 0;
 AntIntCont := 0;
 asm sti end;
end;
```

Ahora necesita el temporizador para sus propios propósitos, pero como muy tarde cuando abandone el programa debería volver a colocar el temporizador en su estado original. Al fin y al cabo, en su coche de alquiler tampoco deja toda la basura tirada. De modo que para comenzar ha de colocar el temporizador en su frecuencia estándar. Esto lo puede obtener volviéndole a indicar al temporizador mediante el valor *36h* en el Port *43h* que va a ser cargada una nueva frecuencia. Después ha de escribir dos veces el valor *0* en el Port *40h*. Con ello el temporizador vuelve a quedar en su estado original. Finalmente debería reponer la rutina original del temporizador. De lo contrario apuntará al Nirvana después de que abandone su programa, lo que habitualmente lleva a un "cuelgue".

```
procedure ApagaTemporizador;
var oldv : pointer;
begin;
  asm cli end;
  port[$43] := $36;
  Port[$40] := 0;
  Port[$40] := 0;
  GetIntVec(OldTimerInt, OldV);
  SetIntVec(8, OldV);
  asm sti end;
end;
```

Ahora que sabemos reprogramar el temporizador, vamos a realizar todo aquello que realmente hemos de hacer en nuestro procedimiento de temporizador. Para comenzar hemos de preocuparnos del antiguo procedimiento del temporizador. Se ha de seguir llamando 18,2 veces por segundo. En el caso de 1050 llamadas de interrupción esto significa que después de cada 58 interrupciones se ha de llamar al antiguo temporizador. Esto se realiza mediante la comparación con la variable

AntIntCont. Como pequeño detalle se va contando el tiempo de reproducción del tema. También esto se realiza en la interrupción. Con 50 llamadas por segundo se ha de incrementar el tiempo del tema cada *0* llamadas del temporizador. Finalmente se llama al procedimiento más importante de la interrupción: el procedimiento que calcula la música. Es necesario el control del retrazado, ya que no es posible una sincronización exacta con el retrazado si durante la espera se calcula música.

```
procedure NuevoTemporizador; interrupt;
var dr : registers;
begin;
 inc(perfcount);

 inc(SecCont);
 inc(AntIntCont);
 if AntIntCont = 58 then begin;
   AntIntCont := 0;
   intr(Oldtimerint, dr);
 end;
 if SecCont = timer_per_second then begin;
   SecCont := 0;
   inc(SongSec);
   if SongSec = 60 then begin;
     inc(SongMin);
     SongSec := 0;
   end;
 end;
 if not in_retrace then calculate_Music;
 Port[$20] := $20;
end;
```

Las rutinas de sonido

Las rutinas para el control de la tarjeta Sound Blaster debería conocerlas del capítulo sobre la programación Sound Blaster. Por ello no vamos a volver a presentarlas todas en este lugar, sino que sólo indicaremos las funciones y los procedimientos más importantes.

procedure wr_dsp_sb16(v : byte);

Mediante este procedimiento puede reproducir un byte en el Port de comandos de la Sound Blaster.

Function SbReadByte : Byte;

Aquí podrá leer un byte de datos de la tarjeta Sound Blaster.

Function Reset_Sb16 : boolean;

Mediante esta función de reinicia la tarjeta Sound Blaster. Si la tarjeta se pudo reiniciar, devuelve el valor *TRUE*, de lo contrario devolverá *FALSE*. Con esta función se puede averiguar la dirección base de la Sound Blaster.

Function Detect_Reg_Sb16 : boolean;M

Esta función se encarga de la obtención de la dirección base, mencionada antes. Inicializa la variable *dsp_adr* con la dirección base de la tarjeta Sound Blaster. Se devuelve *TRUE* cuando se encuentra la dirección base, sino *FALSE*.

Procedure Write_Mixer(Reg,Val : Byte);

Este procedimiento escribe el valor pasado en *val* al registro de mezclador *Reg*. El procedimiento sólo funciona en tarjetas a partir de la Sound Blaster Pro.

Function Read_Mixer(Reg : Byte) : byte;

Mediante esta función se puede leer un valor del registro de mezclador especificado en *Reg*.

Procedure Filtro_Act;

Este procedimiento activa el resaltado de bajos en las tarjetas a partir de la Sound Blaster Pro.

Procedure Filtro_Med;

Para ajustar un sonido normal en las tarjetas a partir de la Sound Blaster Procedimiento, puede emplear este procedimiento.

Procedure Filtro_DesAct;

Puede obtener un resaltado de los agudos mediante la llamada del procedimiento *Filtro_DesAct*.

Procedure Set_Balance(Valor : byte);

Mediante este procedimiento puede ajustar el balance de la salida en las tarjetas estéreo. Para *Valor* se permiten valores entre *0* y *15*. *0* significa a la izquierda y *15* a la derecha.

procedure Set_Volume(Valor : byte);

El volumen con el que se reproduce el archivo MOD se puede ajustar mediante este procedimiento. En las tarjetas a partir de la Sound Blaster Procedimiento, el volumen se ajusta por hardware mediante el mezclador. En la Sound Blaster normal esta regulación se realiza a través de software mediante *outvolume*.

procedure Reset_Mixer;

Al igual que el DSP, también el mezclador se ha de reiniciar a sus valores predeterminados. Para ello se escribe un *0* en el registro de reset del mezclador. Este procedimiento hace esta función.

function Detect_Mixer_sb16 : boolean;

Esta función sirve para la identificación del chip mezclador. Devuelve el valor *TRUE* si se encontró un chip mezclador, y *FALSE* si no fue así. Esta función es importante que se ha de emplear para diferenciar las diferentes tarjetas Sound Blaster. Si la función devuelve el valor *TRUE*, esto significa que hay presente una tarjeta a partir de Sound Blaster Pro.

procedure SbGetDSPVersion;

Para la diferenciación entre Sound Blaster Procedimiento y Sound Blaster 16 necesitará el número de versión de la tarjeta. Si es menor que 4 es una Sound Blaster Procedimiento, sino es Sound Blaster 16.

procedure Set_Timeconst_sb16(tc : byte);

Necesitará esta función para fijar la constante de temporizador de la Sound Blaster. La constante se calcula según la fórmula

```
tc := 256 - (1.000.000 / Frecuencia de sampling).
```

La constante resultante se ha de transmitir mediante el comando *$40*. Al procedimiento ha de pasarle la constante calculada según la fórmula.

```
procedure Exit_Sb16;
```

¡Este procedimiento se ha de llamar al final del programa! Devuelve el valor original a la interrupción desviada de la Sound Blaster y restaura el enmascarado original de interrupciones.

```
procedure Reprod_SB _SbPro _Sb16(Segm,Offs,dsize : word);
```

Mediante este procedimiento se inicia la reproducción del bloque direccionado mediante *Segm* y *Offs*. *Segm* representa el número de la página de memoria y *Offs* la parte de offset restante. El tamaño del bloque se especifica en *dsize*. Según qué tarjeta haya instalada (SB, SB Procedimiento o SB 16), se salta a un procedimiento u otro. Sólo se diferencian en algunas interioridades durante la transferencia DMA. Puede encontrar informaciones exhaustivas en el capítulo 13.

```
procedure dsp_block_sb16(gr : word; bk : pointer; b1,b2 : boolean);
```

Esta función existe para que no sea tan difícil la salida de un bloque mediante DMA. Transmite el bloque direccionado mediante el puntero *bk* con el tamaño *gr* vía DMA. Según si desea reproducir un sample o si necesita el procedimiento para la salida de un MOD, habrá de emplear distintos valores para *b1 y b2*. Si reproduce un MOD, *b1 = TRUE* y *b2 = FALSE*, para un sample o un archivo es justo al revés, es decir, *b1 = FALSE* y *b2 = FALSE*.

Manejo de archivos MOD

La reproducción de archivos MOD se realiza de la siguiente forma: Mediante el procedimiento *periodic_on* se desvía la interrupción de temporizador a una rutina propia y se programa a 50 llamadas por segundo. La rutina direccionada incrementa algunas variables y llama al procedimiento *calculate_music*. Éste comprueba si no se está trabajando en otra rutina, y llama, si no es así, al *Sound_handler*. Éste comprueba si el siguiente bloque a calcular aún no lo está. Si es así, se calcula mediante el procedimiento *inner_proc*. De lo contrario se comprueba si el antiguo bloque calculado ya ha sido transmitido. Si es el caso, se puede calcular un bloque nuevo. Pero para ello primero se han de cargar los nuevos parámetros. Esto lo realizan los procedimientos *nmv_proc* y *Inicializa_mezcla*. Mediante *nmv_proc* se procesa el archivo MOD línea a línea. También regula todos los posibles efectos de sonido.

Ahora que ya conoce el modo genérico de trabajo de la Unit, es hora de echarle un vistazo más de cerca. Para que pueda seguir la Unit con ayuda del código fuente, iremos por orden durante la presentación de los procedimientos, tal y como van apareciendo en la Unit.

El primer procedimiento interesante para nosotros es el procedimiento *get_pctunel*. Obtiene el ancho de paso en el sample en base al tono pasado. El tono es el valor que figura en el archivo MOD. Para obtener el tono, se busca en una matriz que contiene todos los tonos existentes, hasta que se encuentra en valor adecuado. La posición en la tabla de tonos sirve entonces como índice en una tabla con las amplitudes de paso.

```
procedure get_pctunel(altura : word;Var vk : longint);
{
El procedimiento obtiene los decimales y mantisa del tono indicado
(así como se indica en el archivo MOD) para las manipulaciones de
frecuencia.
}
var nct : byte;
   encontrado : boolean;
begin;
 nct := 1;
 encontrado := false;
 while (nct <= 70) and not encontrado do{hasta encontrado o}
 begin;                                  {último valor en la tabla}
   if altura > Octava_MOD[nct] then
   encontrado := true;
   inc(nct);
 end;
 if encontrado then begin;
   vk  := Incfacts[nct-tpw+12];
 end else begin;
   vk  := 0;                             {obtener valores de la tabla}
 end;
end;
```

De manera similar a *get_pctunel* se comporta la función *Nro_Nota*. Se necesita para efectos como los portamentos. Obtiene la posición del tono pasado en la tabla con los valores MOD.

```
function Nro_Nota(altura : word) : integer;
var nct : byte;
   encontrado : boolean;
begin;
 nct := 1;
 encontrado := false;
 while (nct <= 70) and not encontrado do{hasta encontrado}
```

```
  begin;                                    {o último valor en la tabla}
    if altura > Octava_MOD[nct] then
      encontrado := true;
      inc(nct);
    end;
    if encontrado then begin;
      Nro_Nota := nct-1;
    end else begin;
      Nro_Nota := -1;
    end;
  end;
```

A continuación llegamos a dos procedimientos muy importantes. Se trata de *bucle_interno_4* y *bucle_interno_4_stereo*. A través de estos procedimientos se realiza la mezcla en sí de las voces del archivo MOD. Primero se ha de determinar el tamaño a calcular. Ya que siempre calculamos justamente un bloque de reproducción, fijaremos el tamaño en *tamanyobloque*. Ya que trabajamos con Double-Buffering a continuación se ha de determinar el buffer de destino. A continuación se borra el buffer de mezcla. Esto es necesario para no tener los restos de un proceso de mezcla anterior en el buffer. Después se llama el procedimiento de mezcla actual en ensamblador. Éste suma los datos de sampling a la voz correspondiente del buffer de mezcla. Después de que los datos de sampling han llegado al buffer de mezcla de esta forma, se ha de pasar éste al buffer de salida. Para ello primero se han de dividir los valores del buffer de mezcla entre el número de voces. Esto es necesario ya que la Sound Blaster sólo puede reproducir datos de 8 bits. En la Sound Blaster 16 se puede prescindir de este paso, pero los datos se han de pasar entonces como auténticos datos de 16 bits mediante el DMA. Antes de que el procedimiento escriba los datos al buffer de salida, se ha de calcular el volumen de salida actual *outvolume*. Mediante una reducción del volumen de salida puede crear un bonito efecto de Fade Out. Pero veamos primero el código del procedimiento:

```
procedure Bucle_interno_4;
{
 Aquí se realiza la mezcla real de los datos. El búfer se rellena
 con los datos que han sido calculados. Esta es la versión MONO de la
 rutina.
}
begin;
  calc_size := Tam_Bloque;
  if bsw then
    Destino := pt(buffer1)
  else
    Destino := pt(buffer2);

asm
    les di,mixed_data
    mov ax,8080h
```

```
    mov cx,4000
    rept stosw

    mov cx,1
@Bucle_Voces:
    mov mixed_posi,0
    mov Voz_actual,cx
    mov si,cx
    dec si
    shl si,2
    call dword ptr Mixingprocs[si]

    inc cx
    cmp cx,Voces
    jbe @Bucle_Voces

    mov mixed_posi,0
    mov cx,calc_size
@Mixed_2_blk:
    les di,mixed_data
    add di,mixed_posi
    mov ax,es:[di]
    push cx
    mov cx,shiftfactor
    shr ax,cl
    pop cx
    add mixed_posi,2

    mov bx,Destino.sgm          {escribir byte en destino}
    mov es,bx
    mov bx,Destino.ofs
    mul outvolume
    shr ax,6
    mov es:[bx],al
    inc Destino.ofs

    loop @mixed_2_blk
end;
end;
```

El procedimiento *bucle_interno_4_stereo* trabaja prácticamente igual que *bucle_interno_4*, a diferencia de que necesita dos buffers de mezcla, uno para el canal izquierdo y otro para el derecho. Se emplea el ya conocido buffer *mixed_data* para el canal izquierdo y *mixed_data_st* para el canal derecho. La mezcla de las voces funciona análogamente, qué voz se asocia a qué buffer ya se definió en el procedimiento *nmw_proc*. Al transferir los buffers de mezcla al buffer de salida se ha de tener en cuenta que la Sound Blaster necesita alternativamente un valor

para el canal izquierdo y canal derecho. Por ello se han de cargar valores alternadamente de los dos búferes.

```
procedure Bucle_interno_4_stereo;
{
 Aquí se realiza la mezcla real de los datos. El búfer se rellena
 con los datos que han sido calculados. Esta es la versión MONO de la
 rutina.
}
begin;
  calc_size := Tam_Bloque;
  if bsw then
   Destino := pt(buffer1)
  else
   Destino := pt(buffer2);

asm
   les di,mixed_data
   mov ax,8080h
   mov cx,4000
   rep stosw

   les di,mixed_data_st
   mov ax,8080h
   mov cx,4000
   rep stosw

   mov cx,1
@Bucle_Voces:
   mov mixed_posi,0
   mov Voz_actual,cx
   mov si,cx
   dec si
   shl si,2
   call dword ptr Mixingprocs[si]

   inc cx
   cmp cx,Voces
   jbe @Bucle_Voces

   mov mixed_posi,0
   mov cx,calc_size
@Mixed_2_blk:
   les di,mixed_data
   add di,mixed_posi
   mov ax,es:[di]
   push cx
   mov cx,shiftfactor_stereo
```

```
    shr ax,cl
    pop cx

    mov bx,Destino.sgm     {escribir byte a destino}
    mov es,bx
    mov bx,Destino.ofs
    mul outvolume
    shr ax,6
    mov es:[bx],al
    inc Destino.ofs

    les di,mixed_data_st
    add di,mixed_posi
    mov ax,es:[di]
    push cx
    mov cx,shiftfactor_stereo
    shr ax,cl
    pop cx
    add mixed_posi,2

    mov bx,Destino.sgm     {escribir byte a destino}
    mov es,bx
    mov bx,Destino.ofs
    mul outvolume
    shr ax,6
    mov es:[bx],al
    inc Destino.ofs

    loop @mixed_2_blk
  end;
  end;
```

La mezcla en sí se realiza en el módulo de ensamblador de la Unit. Las rutinas externas son imprescindibles, ya que se necesita código del 386 para el Fixpoint-Handling. El procedimiento trabaja de la siguiente manera: en primer lugar se carga el número de la voz actual en el registro *SI*. Sirve como índice en los campos que se refieren a voces. A continuación se carga la variable *calc_size* en el registro *CX*. Describe el número de los datos a calcular.

En un bucle de carga se va comprobando si la voz ha llegado a su final. Si no es el caso, se puede comenzar con la carga del byte de sampling. De lo contrario se ha de comprobar si la voz se repite en un bucle. Si es así, se ha de colocar la posición en el sample al valor inicial de la voz. De otro modo, el sample ha llegado a su final. El procedimiento de mezcla se cambia ahora desde el procedimiento actual al procedimiento vacío *VozVacia*, que no hace otra cosa que volver. Además se pone a cero el volumen de la voz actual y el

procedimiento salta al final del bucle ya que al fin y al cabo no quedan más datos que calcular.

Después de haber realizado la consulta de posición de esta forma, se puede cargar el byte del sample. Para ello se carga el segmento del sample en *ES* y la palabra alta de la posición en la voz en *BX*. Cargamos la palabra alta, ya que la variable de posición es del tipo de coma fija, y en la palabra baja de la misma sólo se encuentra la parte decimal de la posición. Mediante *es:[bx]* se puede cargar el byte del sample en *AL*. En el byte en *AL* se ha de invertir el bit 7, ya que la Sound Blaster sólo reproduce datos *unsigned* (de *0* a *255*), y no como el Amiga o la Gravis Ultrasound datos del tipo *signed* (de *-127* a *128*). A continuación se adapta el volumen de la voz. Para ello se multiplica simplemente el byte en *AL* con el volumen de la voz que se encuentra en *Notvol_Voz*. El resultado de la multiplicación se divide a continuación entre el volumen máximo de *64*. Por suerte se puede sustituir la división por *SHR6*.

Después de haber adaptado el byte de sampling de esta forma, se puede guardar en el buffer de mezcla. Así que primero cargamos la dirección del buffer de mezcla en *es:[di]* y después sumamos la posición en el interior del mismo. Esta posición se encuentra en *mixed_posi*. Esta variable se pone a *0* cada vez, antes de llamar el procedimiento. Ahora se suma en valor del byte calculado al buffer de mezcla (con un *mov* sobrescribiríamos los datos ya existentes).

Finalmente simplemente hay que mover los punteros. Para ello se incrementa *mixed_posi* en 2 y se aumenta la posición en el sample en el valor de coma fija que hay en *IncVal_Voz*.

```
public Voz_normal
Voz_normal proc pascal ;Voz_actual : word;
 pusha
 mov si,Voz_actual
 dec si
 shl si,2  ; para acceso dword

 mov cx,calc_size

@Bucle_carga:

;{ la voz ha llegado al final? }
 mov bx,w Long_Voz[si]
 sub bx,20
 cmp bx,word ptr Posicion_Voz[si+2]
 ja  @Voz_no_al_final
```

```
;{ voz al final, hay bucle? }
  cmp Loop_Long_Voz[si],10
  jae @Voz_en_bucle

;{voz al final y no hay bucle => fuera}
  mov eax,VozVacia
  mov Mixingprocs[si],eax
  mov VolNot_Voz[si],0
  jmp @ende_Voz_normal

; {Parámetro para Stimmel al inicio del bucle}
@Voz_en_bucle:
  mov bx,w Inicio_Bucle_Voz[si]
  mov word ptr Posicion_Voz[si + 2],bx

; {Cargar byte del sample de la Stimmel}
@Voz_no_al_final:
  mov bx,w Segmento_Voz[si]
  mov es,bx
  mov bx,word ptr Posicion_Voz[si + 2]
  mov al,es:[bx]
  sub al,128
  mul b VolNot_Voz[si]
  shr ax,6

@Reprod_voz:
  les di,mixed_data
  add di,mixed_posi
  add es:[di],ax
  add mixed_posi,2

; {Avanzar puntero}
  mov ebx,ValInc_Voz[si]
  add dword ptr Posicion_Voz[si],ebx

  loop @Bucle_carga

@ende_Voz_normal:
  popa
  ret
Voz_normal endp
```

El procedimiento responsable de la mezcla es *calculate_music*. Comprueba si se está ejecutando otro procedimiento actualmente. Si es así, la variable *mycli* tiene el valor *1*. En ese caso se salta directamente al final del procedimiento. Incluso si la variable *music_off* posee un valor distinto de *0*, se termina el procedimiento antes de tiempo. De lo contrario se llama el *Sound_handler*, que es el procedimiento realmente responsable de la mezcla.

Esta rutina comprueba si el siguiente bloque ya ha sido calculado (en este caso *Loop_pos* es mayor que *Speed*) y lo calcula si no es así. De lo contrario se comprueba si el procedimiento de la interrupción Sound Blaster ya ha activado una bandera conforme la salida del bloque antiguo ya ha sido realizada. Sólo entonces se llaman los procedimientos *nmw_proc* e *Inicializa_Mezcla*. Se encargan de que el MOD avance una línea. Si la variable *outfading* vale *TRUE*, se reduce el volumen de salida. En cualquier caso se decrementan los *Desactivación_nota* empleados para efectos como por ejemplo un ecualizador.

```
procedure Sound_handler;
var li : integer;
begin;
 if mycli <> 0 then exit;
 mycli := 1;
 if (Loop_pos > Speed) then begin;
   if phase_2 then begin;
   Nothin_done_count := 0;
   asm
   call [nmw_proc]
   end;
   Inicializa_mezcla;
   Loop_pos := 0;
   phase_2 := false;
   phase_1 := true;
   if outfading then
   if outvolume >= 2 then dec(outvolume,2);
   for li := 1 to 8 do
   if Inicio_Nota[li] > 50 then dec(Inicio_Nota[li],50);
   end;
 end else begin;
   asm call [Proc_interior] end;
   Loop_pos := Speed+2;
 end;
 mycli := 0;
end;

procedure calculate_music; assembler;
asm
 cmp mycli,0
 jne  @Stop_Fin
 cmp  apagar_musica,0
 jne      @Stop_Fin
 pusha
 call Sound_handler
 popa
 @Stop_Fin:
end;
```

Antes de que la mezcla se pueda realizar, las variables se han de cargar primero con los valores adecuados. Para ello se emplea el procedimiento *mezcla_inicio_4* aparte de *nmw_proc*. En él se obtienen primero las amplitudes de paso de las notas a reproducir para todas las voces. Además se cargan los parámetros para el bucle con los valores correspondientes. Además se incrementa o disminuye la amplitud de paso de la voz en correspondencia, si hay un Portamento.

```
procedure inicio_mezcla_4;
var rdiff : real;
   dummy : byte;
   var li : integer;
begin;
 for li := 1 to Voces do begin;
   if note[li] <> 0 then begin;
   Tono_Stimme[li] := (Rm_Song[mli,li,1] and
$0F)*256+Rm_Song[mli,li,2];
   get_pctunel(Tono_Stimme[li],ValInc_Voz[li]);
   end;

   ls[li] := loop_s[In_st[li]];
   ll[li] := loop_l[In_st[li]];
   if ll[li] > 30 then inl[li] := ll[li]+ls[li];
   Loop_Long_Voz[li] := ll[li];
   Inicio_Bucle_Voz[li]  := ls[li];

   case efecto_Voz[li] of
   1 : begin;
       inc(ValInc_Voz[li],Portamento_UP_Voz[li]);
       end;
   2 : begin;
       inc(ValInc_Voz[li],Portamento_do_Stimme[li]);
       end;
   end;
 end;
end;
```

Vamos ahora por el procedimiento que es el principal responsable por el avance en el archivo MOD. Regula el procesamiento por líneas del archivo e inicializa las variables necesitadas por el procedimiento de mezcla. Los efectos que se emplean son tenidos en cuenta por el procedimiento *effect_handling*. Veamos más de cerca esta rutina.

Se le pasan los parámetros el número de la voz a procesar. El procedimiento comprueba si hay presente un efecto en la voz, y si es necesario obtiene el parámetro del efecto. A continuación se comprueba qué efecto está presente procesándolo adecuadamente.

El efecto *01* indica *Portamento up*, *02* por *Portamento down*. En ambos casos primero se obtiene la velocidad con la que se ha de realizar el portamento. A continuación se obtiene el valor de incremento inicial y final del tono correspondiente. Este se divide entre el número de Ticks por línea (*playspeed*). De aquí resulta el valor *Inc* con el que se ha de incrementar o decrementar cada voz por Tick. Mediante el efecto *09* puede regular el offset del sample. Se obtiene mediante la fórmula

```
Posicion := Valor_efecto * 256.
```

Con el efecto *11* puede saltar dentro del arreglo. Para ello simplemente ha de finalizar el patrón actual y leer el número del nuevo valor destino.

El siguiente efecto es muy importante y encuentra aplicación en casi cada tema musical. El número *12* es responsable de determinar el volumen del canal. A *Notvol_Voz* se le asigna el valor que se encuentra en el operando.

El efecto *13* termina el patrón actual y continua la reproducción en el siguiente patrón. Este efecto también se emplea con mucha frecuencia. Simplemente ha de colocar la línea actual al final del archivo MOD, con lo que en la siguiente pasada se carga automáticamente el siguiente patrón.

Detrás del efecto *14* se esconden varios otros. Los *4* bits superiores del operando indican el número del subefecto, y en los *4* bits inferiores se encuentra el operando en sí. En este player sólo se considera el subefecto *12*. Cómo programar los demás efectos lo puede encontrar al final de este capítulo. El efecto *12* tiene el nombre de *Notecut* y termina la reproducción de la voz correspondiente. Para obtener esto, simplemente ha de poner a *0* todos los parámetros que tienen que ver con la reproducción.

El último efecto es el efecto *15*. Sirve para determinar la velocidad. Si el valor del operando es > *15*, se trata de una indicación de Ticks por línea (*Playspeed*). A la variable correspondiente se le puede asignar directamente el valor correspondiente. Si el operando tiene un valor < 15, el valor se refiere al BPM (Beats per minute). Tiene influencia sobre el tamaño del bloque de sampling, que se ha de reproducir por salida. Se calcula según la fórmula

```
Tamaño := Speed * (FrecuenciaSampling div (BPM * 4));
```

En correspondencia, en nuestra Unit se ha de fijar de nuevo el valor para los *Bucles_sonido*. En cualquier caso debería comprobar si los valores determinados están en un rango razonable. Si aparece un valor ilógico en el archivo MOD (lo

que sucede para dificultar el Ripping), su Player se puede "colgar" fácilmente, si el nuevo tamaño sobrepasa al de la memoria reservada.

```
procedure effect_handling(li : integer);
var idx : word;
    Portamento_Speed : word;
    Startnote,
    endnote : word;
    startinc,
    endinc : longint;

begin;
 if Rm_Song[mli,li,3] and $0F <= 15 then begin;
   Eff[li] := 0;
   case (Rm_Song[mli,li,3] and $0F) of
      01 : begin;
             efecto_Voz[li] := 1;
             Portamento_Speed := Rm_Song[mli,li,4];
             Startnote := Nro_Nota(Tono_Stimme[li]);
             Endnote   := Startnote+Portamento_Speed;
             get_pctunel(Octava_MOD[Startnote],Startinc);
             get_pctunel(Octava_MOD[Endnote],Endinc);
             Portamento_UP_Voz[li] := round((Endinc - Startinc) /
playspeed);
           end;
      02 : begin;
             efecto_Voz[li] := 2;
             Portamento_Speed := Rm_Song[mli,li,4];
             Startnote := Nro_Nota(Tono_Stimme[li]);
             Endnote   := Startnote-Portamento_Speed;
             get_pctunel(Octava_MOD[Startnote],Startinc);
             get_pctunel(Octava_MOD[Endnote],Endinc);
             Portamento_do_Stimme[li] := round((Endinc - Startinc) /
playspeed);
           end;
       9 : begin;  { Sample offset }
             Posicion_Voz[li] := longint(Rm_Song[mli,li,4]) shl 24;
           end;
      13 : begin;
             mli := 64;
           end;
      11 : begin;
             mli := 64;
             mlj := Rm_Song[mli,li,4];
           end;
      12 : begin;
             VolNot_Voz[li]  := VolNota(li);
           end;
      14 : begin;
```

```
            case (Rm_Song[mli,li,4] shr 4) of
               12 : begin;
                       inl[li]          := 0;
                       VolNot_Voz[li]   := 0;
                       inp[li]          := 0;
                       Pnk[li]          := 0;
                     end;
            end;
          end;
      15 : begin;
            idx := Rm_Song[mli,li,4];
            if idx <= $f then begin;
            Playspeed := idx;
            Speed := Playspeed*105 div 10;
            Tam_Bloque := Speed * Bucles_Sonido;
          end else begin;
            bpm := idx;
            mod_SetLoop(Frec_Sampling div (BPM * 4));
            Speed := Playspeed*105 div 10;
            Tam_Bloque := Speed * Bucles_Sonido;
          end;
          if Tam_Bloque < 40 then Tam_Bloque := 40;
          if Tam_Bloque > 4000 then Tam_Bloque := 4000;
             end;
    end;
  end;
end;
```

Veamos ahora el procedimiento *nmw_all_4*, que se direcciona en el programa mediante el puntero *nmw_proc*. Comienza con una definición de los procedimientos estéreo. Siempre hay asignadas dos voces al canal izquierdo, y dos al canal derecho. Esto lo puede modificar según sus deseos y sin problemas, empleando simplemente los procedimientos de mezcla según sus propias ideas.

El procedimiento incrementa la primera línea para comenzar. Si esta alcanza un valor mayor de 64, el patrón ha alcanzado el final, ya que sólo hay definidas las líneas *1* hasta *64*. De modo que se ha de posicionar de nuevo la variable de la línea actual en *1*. Simultáneamente se incrementa la posición en los arreglos. Si la nueva posición sobrepasa la longitud del tema, el MOD ha llegado al final y la reproducción comienza desde el principio. De lo contrario se puede determinar el número de nuevo patrón a reproducir. Este patrón se carga en el nuestro temporal, al que siempre accedemos directamente con nuestros procedimientos.

Después de haber corregido la posición en el MOD y cargar el patrón actual, se puede comenzar con el procesamiento de las diferentes voces. Para cada voz se

pone primero a cero la bandera para el efecto actual, ya que los efectos sólo se refieren a una línea en el patrón. A continuación se determina la nota que figura en el MOD, guardándose en la variable *note*. Si esta variable tiene un valor <> 0, se activó una nota y nosotros hemos de seguir con nuestras comprobaciones. Si nos encontramos en el modo estéreo, se determina para la voz el procedimiento de mezcla del campo constante *Stereoprocs*. De lo contrario el procedimiento es siempre la rutina *Voz_normal*.

La variable *Activacion_Nota* se coloca en 500, ya que acaba de activarse una nota. A través de la consulta de esta variable se puede realizar, por ejemplo, un pequeño ecualizador.

La variable *In_St* sirve para el acceso a los valores referidos a instrumentos. Se le asigna el valor de *Note*. Mediante ella se le pasa el puntero al sample a reproducir a la variable *inst*. A continuación se obtiene la longitud de la voz guardándola en la variable *Long_Voz*. La posición en el sample está al principio y por ello es *0*. El volumen del mismo se obtiene mediante el volumen por defecto del sample y se escribe en *Notvol_Voz*. A continuación se guarda el segmento del sample de la voz en *Segmento_Voz*.

Con ello se ha finalizado el Handling de la voz en sí. Sólo se han de tener en cuenta los diferentes efectos. Pero esto ya lo realiza el procedimiento anteriormente descrito *effect_handling* por nosotros.

```
procedure nmw_all_4;
const stereoprocs : array[1..8] of pointer =
  (@Voz_normal,@Voz_normal,@Voz_normal_st,@Voz_normal_st,
   @Voz_normal,@Voz_normal,@Voz_normal_st,@Voz_normal_st);
var idx : byte;
    li : integer;
begin;
   inc(mli);
   if mli > 64 then mli := 1;
   if mli = 1 then begin;
     inc(mlj);
     if mlj > LongTema then begin;
       if mloop then begin;
         mlj := 1;
         move(rm[tema[mlj]] ^,Rm_Song,2048);
       end else begin;
         asm
           call [detener_periodico]
         end;
         apagar_musica := true;
```

```
        MOD_Acabado := true;
      end;
    end else begin;
        move(rm[tema[mlj]] ^,Rm_Song,2048);
    end;
  end;

  for li := 1 to Voces do begin;
    efecto_Voz[li] := 0;
    note[li] := (Rm_Song[mli,li,1] AND $F0)+((Rm_Song[mli,li,3] AND
$F0) shr 4);
    if note[li] <> 0 then begin;
      if stereo then begin;
        Mixingprocs[li] := stereoprocs[li];
      end else begin;
        Mixingprocs[li] := @Voz_normal;
      end;
      Inicio_Nota[li] := 500;
      In_St[li]       := note[li];
      inst[li]:=Ptr(pt(Samp[In_St[li]]).sgm,pt(Samp[In_St[li]]).ofs);
      Long_Voz[li]          := sam_l[In_St[li]];
      Posicion_Voz[li]      := 0;
      VolNot_Voz[li]        := inst_vol[in_St[li]];
      Segmento_Voz[li]      := seg(inst[li]^);
    end;
    effect_handling(li);
  end;
```

Todo el procesamiento MOD no nos sirve de nada, si no tenemos un MOD en memoria. De modo que primero nos hemos de ocupar de cargar un archivo MOD. El procedimiento de carga llamado por el programa principal realiza algunas consultas necesarias, antes de llamar el procedimiento de carga en sí, *init_Song*. Esta rutina carga en la memoria principal el archivo MOD indexado por medio de la variable global *Mod_Name*.

Para comenzar pone a *0* las variables de los instrumentos a reproducir, así como los arreglos completos (*tema*). A continuación se abre el archivo MOD. Si ha ocurrido un error, la rutina se interrumpe con el valor *FALSE*. La rutina ofrece la posibilidad de cargar un archivo MOD desde un archivo grande. Esto tiene especial interés para el empleo en demos. Para poder emplear esta capacidad, primero ha de especificar el tamaño del archivo MOD en bytes en la variable *moddatsize*. Si tiene el valor *0*, se utilizará como tamaño el del archivo abierto. La variable *msp* describe la posición inicial en el archivo MOD (para la posición de comienzo del MOD). Por defecto tiene el valor *0*.

Después de haber abierto correctamente el archivo, se comprueba si se trata de un archivo MOD que pueda procesar. Nuestro player puede reproducir archivos MOD de cuatro y ocho voces. Así que hemos de comprobar si la identificación del archivo MOD contiene uno de los identificadores 'M.K.', 'FLT4' o '8CHN'. Este se encuentra en los archivos de 31 voces en la posición *1080*. Si no coincide la identificación, se trata bien de un archivo MOD de 15 voces, o bien el archivo no es del tipo MOD o carece de identificador.

Ahora que ya se sabe cuántas voces tiene el archivo MOD (31 o 15), se han de leer todas ellas individualmente. Esto se realiza en el bucle de carga de voces. Para comenzar va a la posición correspondiente en el archivo MOD. Allí se lee el tamaño de los samples, de 2 bytes de longitud. EL tamaño del sample es un valor en formato Amiga y se ha de convertir al formato del PC. Para ello se permutan los bytes alto y bajo del valor leído mediante la función *Swap*. A continuación se multiplica el valor con 2, ya que la indicación del archivo MOD se refiere a Words (palabras) y no a bytes.

La variable *Inststart* se decrementa en el tamaño del instrumento. Al principio se cargó con el tamaño del archivo y sirve para determinar el número de patrones. Éste no se ha calculado aún en ninguna parte del MOD. Resulta del tamaño total del archivo, descontando la cabecera y el tamaño de los instrumentos. Ya sólo falta dividir este valor entre el correspondiente tamaño del patrón, es decir, 1024 o 2048 bytes. De esta forma se obtienen la cantidad de patrones definidos.

Después de la especificación de longitud de los samples se lee el volumen por defecto del instrumento. Se trata de un byte que no ha de ser convertido. De lo contrario ocurre de nuevo con los dos valores que hemos de leer a continuación. Tanto el inicio del bucle como la longitud del mismo son palabras de Amiga, que se han de convertir según el procedimiento descrito.

Después de haber leído de esta forma los parámetros básicos para las voces, se comprueba si se trata de un archivo MOD de cuatro u ocho voces. Esta comprobación se realiza mediante una comparación de la identificación MOD con '8CHN'. En caso de una coincidencia se trata de un archivo MOD de 8 voces. En ese caso se coloca la variable *TamPat* a 2048, y a *Voces* se le asigna el valor *8*. De lo contrario, se emplean los valores *1024* y *4*. Según si el archivo se ha de reproducir en mono o en estéreo, se colocan a continuación los punteros para los procedimientos de control de mezcla a la versión mono o estéreo de la rutina.

Después de que esto haya ocurrido, se cargan los samples. Primero se reserva el espacio de memoria necesario para todos los instrumentos. A continuación se cargan los datos de sampling del disco a la memoria.

El último paso importante lo representa la carga de los patrones. Para ello averigüe primero el número de patrones del archivo MOD. Puede cargar ahora los patrones en un bucle. Tenga en cuenta que un patrón de ocho voces necesita el doble de memoria que uno de cuatro voces. Ahora sólo le queda cargar el arreglo de 128 bytes y la longitud del tema. El arreglo lo puede encontrar en la posición de offset *952* y la longitud del tema en la posición *950*.

Bien, ya lo ha conseguido: el archivo MOD se encuentra en la memoria. Para que todo esté completo aún debería cargar los nombres del archivo MOD y de los instrumentos. No son necesarios para la reproducción, pero deberían visualizarse en el player. El nombre del tema se encuentra directamente al principio del archivo, y los nombres de los samples siempre al principio de las informaciones de sample. Los nombres se guardan como cadenas terminadas en *0*. Es decir, que su último carácter tiene el valor *#0*. Ahora hay que convertir estas cadenas al formato de Pascal. Esto se puede realizar con la función *ConvertString*. Si la carga tuvo éxito, la función aquí presentada termina con el valor *TRUE*. Las variables para la línea actual y las posiciones en el patrón se colocan ambas al principio.

```
FUNCTION ConvertString(Source : Pointer; Size : BYTE):String;
VAR
   WorkStr : String;
BEGIN
   Move(Source^,WorkStr[1],Size);
   WorkStr[0] := CHR(Size);
   ConvertString := WorkStr;
END;

function init_Song : boolean;
const kenn1 : string = 'FLT4';
      kenn2 : string = 'M.K.';
      kenn3 : string = '8CHN';
var rmod : file;
    sgr : word;                    {tamaño de un sample                }
    inststart : longint; {posición en archivo, donde comienzan datos}
    datgr : longint;               {tamaño del archivo MOD             }
    Mkg : array[1..4] of char;     {para reconocimiento MOD            }
    pAyuda : ^byte;
    strptr : pointer;
    Id_Canal : array[1..4] of char;
    Cadena_Id : string;
    instancia : byte;
    idx : integer;
```

```
begin;
 In_St[1] := 0;
 In_St[2] := 0;
 In_St[3] := 0;
 In_St[4] := 0;
 In_St[5] := 0;
 In_St[6] := 0;
 In_St[7] := 0;
 In_St[8] := 0;
 for mlj := 0 to 128 do
   Lied[mlj] := 0;
 {$I-}
 assign(rmod,Mod_Name);
 reset(rmod,1);
 {$I+}
 if IOresult <> 0 then begin;
   init_song := false;
   exit;
 end;
 if Tam_Arch_MOD <> 0 then datgr := Tam_Arch_MOD else
 datgr := filesize(rmod);
 inststart := datgr;
 seek(rmod,1080);
 blockread(rmod,Id_Canal,4);
 Cadena_Id := Id_Canal;
 if (Cadena_Id <> kenn1) and (Cadena_Id <> kenn2)
 and (Cadena_Id <> kenn3) then begin;
   instancia := 15;
 end else begin;
   instancia := 31;
 end;

 if instancia = 31 then begin; {obtenidas 31 voces por identificador}
 for mlj := 1 to 31 do begin;
  idx := mlj;
  seek(rmod,msp+42+(idx-1)*30);
  blockread(rmod,sgr,2);
  sgr := swap(sgr) * 2;
  if sgr <> 0 then inststart := inststart - sgr;
  Sam_l[idx] := sgr;
  seek(rmod,msp+45+(idx-1)*30);
  blockread(rmod,inst_vol[idx],1);
  blockread(rmod,loop_s[idx],2);
  blockread(rmod,loop_l[idx],2);
  loop_s[idx] := swap(loop_s[idx])*2;
  loop_l[idx] := swap(loop_l[idx])*2;
 end;

 seek(rmod,msp+1080);
```

```
blockread(rmod,Mkg,4);
if pos('8CHN',Mkg) <> 0 then begin;
  Tam_Patron     := 2048;
  Voces          := 8;
  shiftfactor          := 3;
  shiftfactor_stereo := 3;
end else begin;
{ Archivo Mod de cuatro voces }
  Tam_Patron     := 1024;
  Voces          := 4;
  shiftfactor          := 2;
  shiftfactor_stereo := 2;
end;
Proc_mezcla := @inicio_mezcla_4;
nmw_Proc       := @nmw_all_4;
if stereo then
  Proc_interior      := @Bucle_interno_4_stereo
else
  Proc_interior      := @Bucle_interno_4;


seek(rmod,msp+inststart);
for mlj := 1 to 31 do begin;
 idx := mlj;
 getmem(Samp[idx],Sam_l[idx]);
 blockread(rmod,Samp[idx]^,sam_l[idx]);
end;

datgr := inststart - 1083;
Num_pat := datgr div Tam_Patron;
for mlj := 0 to Num_pat-1 do begin;
  getmem(rm[mlj],2048);
  fillchar(rm[mlj]^,2048,0);
  seek(rmod,msp+1084+mlj*Tam_Patron);
  pAyuda := ptr(seg(rm[mlj]^),ofs(rm[mlj]^));
  for mli := 0 to 63 do begin;
    pAyuda := ptr(seg(rm[mlj]^),ofs(rm[mlj]^)+mli*32);
   blockread(rmod,pAyuda^,Tam_Patron div 64);
  end;
end;
seek(rmod,msp+952);
blockread(rmod,Lied,128);

getmem(strptr,25);
for i := 0 to 30 do begin;
  seek(rmod,msp+20+i*30);
  blockread(rmod,strptr^,22);
  NombresInst[i+1] := convertstring(strptr,22);
end;
```

```
seek(rmod,msp);
blockread(rmod,strptr^,20);
songname := convertstring(strptr,20);
freemem(strptr,25);

seek(rmod,msp+950); { de 470}
blockread(rmod,LongTema,1);
end else begin;
for mlj := 1 to 15 do begin;
 seek(rmod,msp+42+(mlj-1)*30);
 blockread(rmod,sgr,2);
 sgr := swap(sgr) * 2;
 if sgr <> 0 then inststart := inststart - sgr;
 Sam_l[mlj] := sgr;
 seek(rmod,msp+45+(mlj-1)*30);
 blockread(rmod,inst_vol[mlj],1);
 blockread(rmod,loop_s[mlj],2);
 blockread(rmod,loop_l[mlj],2);
 loop_s[mlj] := swap(loop_s[mlj])*2;
 loop_l[mlj] := swap(loop_l[mlj])*2;
end;

for mlj := 16 to 31 do begin;
 Sam_l[mlj] := 0;
 loop_s[mlj] := 0;
 loop_l[mlj] := 0;
end;

if pos('8CHN',Mkg) <> 0 then begin;
  Tam_Patron     := 2048;
  Voces          := 8;
  shiftfactor          := 3;
  shiftfactor_stereo := 3;
end else begin;
{ archivo MOD de 4 voces}
  Tam_Patron     := 1024;
  Voces          := 4;
  shiftfactor          := 2;
  shiftfactor_stereo := 2;
end;
Proc_mezcla := @inicio_mezcla_4;
nmw_Proc       := @nmw_all_4;
if stereo then
  Proc_interior      := @Bucle_interno_4_stereo
else
  Proc_interior      := @Bucle_interno_4;
seek(rmod,msp+inststart);
for mlj := 1 to 15 do begin;
```

```
    getmem(Samp[mlj],Sam_l[mlj]);
    blockread(rmod,Samp[mlj]^,sam_l[mlj]);
  end;

  datgr := inststart - 603;
  Num_pat := datgr div Tam_Patron;
  for mlj := 0 to Num_pat-1 do begin;
    getmem(rm[mlj],2048);
    fillchar(rm[mlj]^,2048,0);
    seek(rmod,msp+1084+mlj*Tam_Patron);
    pAyuda := ptr(seg(rm[mlj]^),ofs(rm[mlj]^));
    for mli := 0 to 63 do begin;
      pAyuda := ptr(seg(rm[mlj]^),ofs(rm[mlj]^)+mli*32);
      blockread(rmod,pAyuda^,Tam_Patron div 64);
    end;
  end;
  seek(rmod,msp+472);
  blockread(rmod,Lied,128);

  getmem(strptr,25);
  for i := 0 to 14 do begin;
    seek(rmod,msp+20+i*30);
    blockread(rmod,strptr^,22);
    NombresInst[i+1] := convertstring(strptr,22);
  end;

  for i := 15 to 30 do begin;
    NombresInst[i+1] := '';
  end;
  seek(rmod,msp);
  blockread(rmod,strptr^,20);
  songname := convertstring(strptr,20);
  freemem(strptr,25);

  seek(rmod,msp+470);
  blockread(rmod,LongTema,1);
  end;


  mlj := 0;
  mli := 0;
  close(rmod);
  init_song := true;
end;
```

Para cargar el archivo MOD no sólo se ha de realizar la carga en sí. También se han de inicializar toda una serie de otras variables. De todo esto se encarga la función *carga_archivomod*. Como primer parámetro se le ha de pasar el nombre del archivo MOD que desea cargar, en caso necesario con la vía de acceso completa. Para el

segundo y tercer parámetro simplemente indique *AUTO*. El último parámetro determina la frecuencia de reproducción. Si no dispone precisamente de un 386SX de 16 MHz, puede indicar tranquilamente el valor 22. El archivo MOD se reproducirá con 22 kHz. De lo contrario aún se deberían obtener resultados aceptables con 16 (kHz).

Lo primero que comprueba la función es si el nombre de archivo el válido. Si no es así, interrumpe con el mensaje de error *-1*. A continuación se comprueba si la salida se ha de realizar en mono (*force_mono = TRUE*). En este caso *Stereo* se coloca en *FALSE*. Si hay instalada una Sound Blaster Pro, se ha de activar el modo de reproducción correcto en esta tarjeta. Esto sólo es así en la SB Procedimiento y se debe a la capacidad de los desarrolladores de Creative Labs. Después de haber terminado los tratamientos de estéreo sólo quedan por inicializar algunos datos. Esto nos los realiza el procedimiento *init_data*.

Mediante *init_song* se carga el archivo MOD en la memoria. A continuación se fijan la frecuencia y la velocidad estándar de reproducción. Después se ha de enviar la tasa de sampling correspondiente a la Sound Blaster y activar el altavoz. Si hay presente una Sound Blaster con capacidad de estéreo, se coloca el balance en el centro y se ajusta el volumen almacenado en *Mastervolume*. El procedimiento termina ahora con un resultado de función de *0*.

```
function carga_archivo_MOD(modname : string;ispeed,iloop :
integer;freq : byte) : integer;
var df : file;
    sterreg : byte;
    fgr : longint;
begin;
 PLAYING_MOD := true;
 PLAYING_VOC := false;
 outfading := false;
 outvolume := 63;
 Mod_Name := modname;
 {$I-}
 assign(df,Mod_name);
 reset(df,1);
 {$I+}
 if IOResult <> 0 then begin;
   {$I-}
   close(df);
   carga_archivo_MOD := -1;              { ¡archivo no se encontró!   }
   exit;
end;
{$I-}
fgr := filesize(df);
close(df);
music_played := true;
```

```
  apagar_musica := false;
  MOD_Finalizado := false;

  if ispeed <> AUTO then Speed3 := ispeed;
  if iloop <> AUTO then Loop3  := iloop;
  if force_mono then stereo := false;
  if force_sb then begin;
    if Sbl6Detected then stereo := false;
    Sbl6Detected := false;
  end;

  if SBProdetected then begin;
    if stereo then begin;
      sterreg := Read_Mixer($0e);
      write_Mixer($0e,sterreg OR 2);
    end else begin;
      sterreg := Read_Mixer($0e);
      write_Mixer($0e,sterreg AND $FD);
    end;
  end;
  init_data;
  if init_song then begin;
    phase_1 := false;
    phase_2 := true;
    mycli := 0;

    mod_Samplefreq(freq);
    Playspeed    := 6;
    Speed := Playspeed*105 div 10;
    bpm := 125;
    mod_SetLoop(Frec_Sampling div (BPM * 4));
    Tam_Bloque := Speed * Bucles_Sonido;
    if Tam_Bloque < 100 then Tam_Bloque := 100;
    if Tam_Bloque > 4000 then Tam_Bloque := 4000;

    asm call [nmw_proc] end;
    set_timeconst_sbl6(Sampling_Rate);
    Inicializa_mezcla;
    SegReprod := 0;
    MinReprod := 0;
    wr_dsp_sbl6($D1);
    if sbl6detected or sbprodetected then begin;
       filter_Mid;
      Set_Balance(Balance);
      Set_Volume(Mastervolume);
    end;
    Carga_Arch_MOD := 0;
  end else begin;
    Carga_Arch_MOD := -3;  { error al cargar un tema     }
  end;
end;
```

Finalmente ha de saber cómo iniciar realmente el archivo MOD. La llamada es muy sencilla, concretamente mediante el procedimiento *periodic_on*. Si se llamó a esta rutina, comienza inmediatamente la reproducción del archivo MOD. El procedimiento coloca la bandera *ultima_salida* en *FALSE*, ya que estamos comenzando la salida y no queremos terminarla. A continuación se reproduce el bloque calculado mediante DMA. A través de la inversión de la variable booleana *bsw* puede pasar al siguiente bloque del Double-Buffering. Ahora sólo se han de cargar los valores para la siguiente línea, lo que realizan a su vez los procedimientos *nmw_proc* e *Inicializa_mezcla*. El último paso es casi el más importante: la interrupción de temporizador se ha de desviar a un procedimiento propio y su frecuencia se ha de fijar de nuevo. Esto lo puede realizar mediante el procedimiento *AjustaTemporizador*. A esta rutina se le ha de pasar como primer parámetro el procedimiento de interrupción *NuevoTemporizador*, que ahora veremos en detalle. Como segundo parámetro la rutina necesita la frecuencia de interrupciones por segundo. Aquí puede especificar la constante *timer_per_second*, que tiene el valor *50*. Y eso ha sido todo. El archivo MOD ahora debería estar sonando alegremente, una vez haya llamado el procedimiento.

```
Procedure activar_periodic;
Begin
 outvolume := 64;
 Ultima_Reprod := false;
{ for Loop_pos := 1 to Speed do begin;}
    asm call [Proc_interior] end;
{ end;}
 dsp_block_sb16(Tam_Bloque,Tam_Bloque,buffer1,true,false);
 bsw := not bsw;
 Loop_pos := 0;
 asm
   call [nmw_proc]
end;
 Inicializa_mezcla;

 init_sbperiod(@desactivar_periodic);
 music_played := true;
 Ajusta_Temporizador(@NeuerTimer,timer_per_second);
End;
```

Ahora ha iniciado el archivo MOD, pero en algún momento seguramente querrá finalizar la reproducción. Esto se resuelve de la misma forma sencilla que el inicio de la reproducción del archivo MOD. Simplemente ha de llamar al procedimiento *periodic_off*, y ya termina la salida del archivo MOD. El procedimiento coloca la variable *ultima_salida* a *TRUE*, lo que provoca que en la rutina de interrupción no se especifica ningún vbloque nuevo. A continuación volvemos a apagar la interrupción del temporizador. Es decir, lo restauramos a sus valores antiguos. Esto nos lo realiza el procedimiento *ApagaTemporizador*.

```
Procedure periodic_off;
Begin
  ultima_Salida := true;
  ApagaTemporizador;
End;
```

Ahora que ya puede reproducir archivos MOD con esta Unit, vamos a por un pequeño bombón, la reproducción de archivos VOC. Para ello sólo necesita dos procedimientos: *Init_Voc*, con la que comienza la reproducción y *Voc_Done*, que termina la salida. Primero veamos la rutina *Init_Voc* un poco más de cerca.

Al procedimiento se le ha de pasar como parámetro el nombre del archivo VOC a reproducir, si es necesario con la vía de acceso completa. Esta rutina carga las banderas *Playing_MOD* con *FALSE* y *Playing_VOC* con *TRUE*. Con ello se le da a entender al procedimiento de interrupción que ahora se está reproduciendo un archivo VOC y no uno MOD. Por defecto se desactiva la reproducción estéreo, aunque un archivo VOC de este tipo la vuelve a activar. A continuación se abre el archivo VOC y se lee la cabecera. Si el identificador que se encuentra en la cabecera no es *'Creative Voice File' + #1A*, no se trata de un archivo VOC, o al menos de uno incorrecto.

A continuación se lee el primer bloque Voice del archivo VOC. Su identificación se encuentra en la posición 2 del campo leído. Hay tres bloques Voice que nos puedan interesar. Por una parte está el bloque *1*. Es el tipo de bloque más sencillo. De él se puede leer directamente la tasa de sampling del mismo. Se encuentra en la posición *6* y puede ser enviada directamente al DSP.

Mucho más incómodo es el tipo de bloque *8*. Aquí han vuelto a atacar los geniales ingenieros de Creative Labs. En vez de guardar la verdadera frecuencia del sample, se amplió la constante de sampling interna de 8 a 16 bits, así que primero se ha de leer esta tasa de sampling. Se puede calcular con la fórmula

```
Frecuencia := 256000000 div (65536 - SR)
```

la verdadera frecuencia del sample. Ahora se ha de comprobar si se trata de un sample estéreo. Si es así, se ha de volver a activar la salida estéreo y dividir la frecuencia de sampling entre *dos*. De la frecuencia de sample obtenida así se puede calcular con la fórmula

```
SB_SR := 256 - (10000000 DIV Frecuencia_de_sampling)
```

la frecuencia que se le ha de pasar a la Sound Blaster. Finalmente se ha de mencionar el tipo de Voice *9*. Este tipo se introdujo junto con la Sound Blaster 16. Creative Labs parece seguir la filosofía "Para cada tarjeta nueva, un formato propio, para

una buena compatibilidad...". En este formato por fin se guarda la verdadera frecuencia de sampling en el archivo. Se encuentra en los bytes *6* y *7*. El byte *11* tiene el valor 2 si hay presente un sample estéreo. En este caso se ha de volver a activar de nuevo la reproducción estéreo. A continuación se ha de comprobar si se dispone de una Sound Blaster Procedimiento (¡o sólo una SB Pro!). Si es así se ha de multiplicar la frecuencia de sampling. De lo contrario se puede adoptar directamente. La frecuencia a indicarle a la Sound Blaster se vuelve a calcular según la fórmula:

```
SB_SR := 256 - (10000000 DIV Frecuencia_de_Sampling)
```

En el caso de un bloque mono se puede obtener la frecuencia directamente según la fórmula. Después de haberle "arrancado" al archivo VOC su frecuencia de sampling, sólo queda transmitirla a la Sound Blaster. Esto se realiza mediante el procedimiento *set_timeconst_sb16*. A él sólo le hemos de indicar la frecuencia que acabamos de determinar.

A continuación hemos de leer el primer bloque de los datos de sampling del archivo VOC. Los bloques tienen un tamaño estándar de 2500 bytes. El primero se lee en el buffer *Buffer1*. Si el tamaño restante de archivo es mayor que el tamaño del buffer, también se lee el segundo bloque. Se guardará en el *Buffer2*.

Después se ha de activar el altavoz. Esto es necesario cada vez antes de comenzar la reproducción de un archivo VOC, ya que no podemos presuponer un altavoz conectado. Si no hay ninguna Sound Blaster 16 instalada en el ordenador, ahora se ha de reprogramar el chip mezclador para la salida mono y estéreo. Después de haber superado esta barrera Creative Labs, se puede iniciar la reproducción del bloque mediante el procedimiento *dsp_block_sb16*. Todo lo demás lo maneja la interrupción Sound Blaster.

Para finalizar la salida del archivo VOC ha de llamar al procedimiento *Voc_Done*. Carga la variable *lastone* con *TRUE* y de esta manera le indica al procedimiento de interrupción que no se ha de reproducir ningún bloque más. Ahora se puede volver cerrar el archivo VOC y reiniciar la tarjeta Sound Blaster. El reset es importante para que otras rutinas Sound Blaster vuelvan a encontrar a la tarjeta en su estado inicial.

```
procedure Init_Voc(filename : string);
const VOC_Id : string = 'Creative Voice File'+#$1A;
var ch : char;
    Cadena_Id : string;
    ct : byte;
    h : byte;
    error : integer;
    srlo,srhi : byte;
```

```
    SR : word;
    Samplingr : word;
    stereoreg : byte;
  begin;
    PLAYING_MOD := false;
    PLAYING_VOC := true;
    VOC_READY   := false;
    vocsstereo := stereo;
    stereo := false;

    assign(vocf,filename);
    reset(vocf,1);
    if filesize(vocf) < 5000 then begin;
      VOC_READY   := true;
      exit;
    end;
    blockread(vocf,voch,$19);
    Cadena_Id := voch.Cadena_Id;
    if Cadena_Id <> VOC_Id then begin;
      VOC_READY   := true;
      exit;
    end;

    Blockread(vocf,inread,20);
    vblock.Identificador := inread[2];

    if vblock.Identificador = 1 then begin;
      vblock.SR := inread[6];
    end;

    if vblock.Identificador = 8 then begin;
      SR := inread[6]+(inread[7]*256);
      Samplingr := 256000000 div (65536 - SR);
      if inread[9] = 1 then begin; {stereo}
        if sb16detected then samplingr := samplingr shr 1;
        stereo := true;
      end;
      vblock.SR := 256 - longint(1000000 DIV samplingr);
    end;

    if vblock.Identificador = 9 then begin;
      Samplingr := inread[6]+(inread[7]*256);
      if inread[11] = 2 then begin; {stereo}
        stereo := true;
        if sbprodetected then samplingr := samplingr * 2;
        vblock.SR := 256 - longint(1000000 DIV (samplingr));
      end else begin;
        vblock.SR := 256 - longint(1000000 DIV samplingr);
      end;
    end;
```

```
    if vblock.SR < 130 then vblock.SR := 166;

    set_timeconst_sb16(vblock.SR);

    TamBloc := filesize(vocf) - 31;
    if TamBloc > 2500 then TamBloc := 2500;
    blockread(vocf,vocb1^,TamBloc);

    ch := #0;
    fgr := filesize(vocf) - 32;
    fgr := fgr - TamBloc;
    Bloque_Activo := 1;

    if fgr > 1 then begin;
      blockread(vocf,vocb2^,TamBloc);
      fgr := fgr - TamBloc;
    end;

    wr_dsp_sb16($D1);
  lastone := false;

  if not sb16Detected then begin;
    if Stereo then begin;
      stereoreg := Read_Mixer($0E);
      stereoreg := stereoreg OR 2;
      Write_Mixer($0E,stereoreg);
    end else begin;
      stereoreg := Read_Mixer($0E);
      stereoreg := stereoreg AND $FD;
      Write_Mixer($0E,stereoreg);
    end;
  end;
  pause_voc := false;
  dsp_block_sb16(TamBloc,TamBloc,vocb1,false,true);
end;
procedure voc_done;
var h : byte;
begin;
  lastone := true;
{ repeat until dsp_rdy_sb16;}
  close(vocf);
  Reset_Sb16;
  stereo := vocsstereo;
end;
```

Finalmente llegamos a un procedimiento de significado central, el procedimiento de interrupción Sound Blaster. Este procedimiento se ejecuta cada vez que la Sound Blaster avisa de que se ha terminado una transferencia DMA. Lo primero que comprueba la rutina es si se trata de una comprobación de interrupción para el reco-

nocimiento por hardware de la Sound Blaster. En ese caso se coloca la bandera *IRQDetected* en *TRUE*. La rutina de detección sabe entonces que se ha encontrado la interrupción adecuada.

Si por el contrario se trata de una interrupción normal, se diferencia si la Unit actualmente está ocupada con la reproducción de un MOD o un VOC. Si se reproduce un MOD, primero se lee un byte de port de datos de la Sound Blaster. Esto es necesario para que se pueda volver a acceder a la Sound Blaster. A continuación se carga la variable *dsp_rdy_sb16* con *TRUE*. Esto indica que el bloque ha sido reproducido hasta el final. Ahora se comprueba si se ha de continuar la salida. Esto es así cuando *ultima_salida* tiene el valor FALSE. En ese caso se inicia la reproducción del bloque actualmente activo mediante una llamada de *dsp_block_sb16*. Finalmente se activa una bandera, de modo que el procedimiento del temporizador puede avanzar una línea en el MOD.

Cuando se reproduce un archivo VOC, el procedimiento es algo distinto. También aquí se ha de realizar primero un acceso de lectura a la Sound Blaster. A continuación se comprueba sin embargo si el archivo VOC no ha llegado al final y si ha de seguir reproduciéndose. Si es así, se envía el bloque actualmente activo a la Sound Blaster mediante el procedimiento *dsp_block_sb16*. A continuación se cargan nuevos datos de sampling en el otro buffer desde el archivo VOC. Después se convierte el bloque actualmente pasivo en activo y viceversa. Si el archivo VOC hubiera llegado al final y no se deseara ninguna reproducción más, se pone *dsp_rdy_sb16* en *TRUE* y se desactiva el altavoz. Finalmente se coloca en *TRUE* la variable booleana *VOC_READY*. Tenga en cuenta que al final del procedimiento de interrupción ha de escribir el valor *20h* en el port *20h*, ya que si no no se ejecutarán más interrupciones.

Consejos para la programación de efectos

En la Unit presentada aún no están implementados todos los efectos Protracker. Los incluidos son suficiente como para reproducir aproximadamente el 80 por ciento de los archivos MOD. Si quiere ampliar los restantes efectos, o bien quiere escribir su propio player, se encontrará ante el siguiente problema: ¿Cómo se programan estos efectos?

Ya que hay una auténtica laguna informativa en cuanto a este tema, vamos a presentarles a continuación todos los efectos con un principio de solución. Cuando hablemos del parámetro x queremos significar los cuatro bits superiores del operando. El parámetro y designa los cuatro bits inferiores del operando. Así que xy representa a todo el operando completo. Comencemos con la enumeración de los efectos con:

Arpeggio

En el *Arpeggio* se reproduce el sonido en tres tonos diferentes en rápida secuencia. Los tonos son por una parte el tono original, después el *tono + parámetro x* y el *tono + parámetro y*. Para cada tick ha de activar la siguiente de las frecuencias de los tonos y después puede calcular la voz con normalidad.

Portamento up/Portamento down

También el efecto de *Portamento* es bastante fácil de programar. En cada Tick se ha de restar o sumar el valor especificado en xy de la amplitud de paso. Por lo demás, el cálculo es normal.

Portamento to Note

El efecto *Portamento to Note* requiere algo más de trabajo, pero en principio funciona como un *Portamento up/down*. Cuando se emplea este efecto, no se considera la nota indicada en la línea como nota a reproducir, sino como nota de destino del Portamento. Primero se ha de comprobar si se ha de emplear un *Portamento up* o un *Portamento down*. A continuación se incrementa o disminuye el paso de la nota con cada Tick en el valor especificado en xy. De esta forma, xy describe la velocidad del Portamento. Este se sigue aplicando hasta que se alcanza la nota de destino. Si no se pudo alcanzar en una línea, se continúa el Portamento en la siguiente, siempre y cuando no se haya especificado otro efecto. Si el campo de notas de esta línea está vacío, se emplea como nota de destino la última nota indicada. De lo contrario será la nota que se encuentra en la línea.

Vibrato

El efecto de Vibrato es bastante complejo. Para él se necesitan tres tablas para los tres tipos de onda posibles. Estas formas de onda son una onda senoidal completamente normal, una onda cuadrada y una onda de diente de sierra, llamada "Ramp down". Las entradas en las tablas se han de encontrar entre -*255* y *255*. La media oscilación, es decir la distancia entre dos puntos de intersección de la curva de la onda en el eje y, ha de tener 32 entradas. De modo que la onda completa se compone de 64 entradas. En cada tick se ha de incrementar ahora la posición del índice en la tabla con el parámetro x. El valor obtenido de la tabla mediante el índice se ha de multiplicar con el parámetro y, dividiendo el resultado entre *256*. El valor resultante se suma a la amplitud de paso de la nota. Si el campo xy contiene el valor *0*, ha de emplear el último valor de Vibrato utilizado. El Vibrato se mantiene hasta que se comienza una nueva nota. Tenga en cuenta que no se resetea el Vibrato cuando se pasa a la siguiente línea del tema.

Portamento y Volume sliding

La combinación de estos dos efectos se puede dividir con facilidad en dos efectos individuales, tratándolos por separado. En cada tick ha de reducir primero el volumen de la voz de la forma correspondiente y a continuación, como descrito anteriormente, modificar el tono de la voz.

Vibrato y Volume sliding

En esta combinación de efectos el Vibrato simplemente se continúa y el volumen se incrementa o reduce de la forma adecuada.

Tremolo

Este efecto está estrechamente emparentado con el efecto de Vibrato. La única diferencia es que aquí no se modifica la velocidad de reproducción de la voz, sino su volumen. Este efecto se denomina frecuentemente como Vibrato de volumen y es una manera legítima de sustituir un Vibrato. Para su realización puede emplear el mismo método que en el Vibrato, incluso las mismas tablas. Sólo tenga en cuenta de no sobrepasar los valores máximos para el volumen, al modificar éste.

Volume Sliding

El *Volume Sliding* puede realizarse en *up* o *down*. Si se emplea el parámetro x, se trata de un incremento del volumen, en caso de emplear el parámetro y es una reducción. En cada tick se modifica el volumen en el valor indicado en el parámetro, hasta que la línea se ha terminado o se ha llegado a un valor límite del volumen.

Position Jump

Este efecto se maneja con facilidad. Con él puede saltar a la posición del arreglo indicada en xy. Esto no debería representar ningún problema, ya que simplemente ha de saltar al final del patrón actual, y allí asignarle el valor de xy a la variable de la posición en el arreglo.

Set Volume

El significado de este efecto ha llevado a grandes discusiones en los círculos especializados. Simplemente fije el volumen especificado en xy para la voz correspondiente. ¡Ya está!

Pattern Break

Este efecto también se puede realizar con sencillez. Sin embargo no es reproducido correctamente por todos los players de MOD (por ejemplo DMP 2.92). Para comenzar, ha de finalizar el patrón actual. Habitualmente esto siempre se soporta,

aunque se ignore el significado del patrón xy. Normalmente tiene el valor *0* (en más del 90 por ciento de todos los MOD). Si no fuera así, designa la línea del siguiente patrón en la que se ha de comenzar. En ese caso no ha de saltar al inicio del patrón, sino a esta línea.

Set Speed

Este efecto sirve la fijar la velocidad de reproducción MOD. El valor estándar está en 125 Beats per minute (BMP) con una velocidad de 6 ticks. Si xy tiene un valor de hasta *0Fh*, el dato de velocidad se refiere a esta (número de ticks). De lo contrario el valor indica los BPM a emplear.

Ahora que ya conoce los efectos normales, vamos a conocer los efectos ampliados. Ya sabemos que son "puro lujo", pero no deberían faltar en ningún player que se precie.

Set Filter

Este es el efecto más sencillo de programar. Ya que es específico del hardware del Amiga, simplemente se ignora.

Fine Portamento up/down

Este efecto funciona exactamente igual que un Portamento normal. Sin embargo no se modifica el tono en cada tick, sino una sola vez durante la inicialización.

Glissando

Puede programar un *Glissando* de la siguiente manera: genere una tabla con todas las notas enteras posible. Ejecute su Portamento con toda normalidad. Ahora ha de redondear el valor así obtenido con al siguiente valor de la tabla.

Set Vibrato Waveform

Mediante este efecto puede seleccionar una de las tres posibles tablas para el Vibrato. La mejor forma de direccionar la tabla actual es mediante un puntero. En ese caso sólo habrá de colocar el puntero a la dirección de la tabla seleccionada, para realizar este efecto.

Set Loop

Guárdese la posición actual (línea) en el patrón. A este punto se volverá de nuevo en el bucle.

Jump to Loop

Jump to Loop es la segunda parte del comando de bucle. Con él salta a la posición determinada por *Set Loop*. Esto se repite tantas veces como se indica en y.

Set Tremolo Waveform

Aquí se aplica lo mismo que en *Set Vibrato Waveform*.

Retrigger Note

Si el tick actual es igual a parámetro y, se ha de cargar la posición en el sample con la posición inicial.

Fine Volume slide up/down

Aumente o disminuya el volumen de la voz durante la inicialización con el valor indicado en y. No se realiza ningún Update durante los ticks.

Note Cut

Mediante este comando detiene la reproducción de una voz. El parámetro y indica el tick en el que la voz ha de finalizar. Para terminar la voz, sólo ha de modificar la posición en la voz, o la longitud de la misma.

Note Delay

El parámetro y indica el valor por el cual se ha de retardar la reproducción de la voz. La indicación se realiza en ticks.

Pattern Delay

Este efecto es parecido a *Note Delay*. Sólo que aquí no se retarda la reproducción de una sola voz, sino de todas las voces en la cantidad indicada en ticks.

Invert Loop

Durante la interpretación de este efecto hemos de apoyarnos en suposiciones. Suponemos que el patrón se reproduce de atrás adelante, desde la posición actual hasta la señalada por *Set Loop*. Pero no hemos encontrado ningún MOD en el que se haya empleado este comando. Por ello prácticamente ningún player lo soporta.

El Mod-Player MOD386

La mejor forma de ver el empleo de la Unit *MOD_SB*, lo podrá ver en el pequeño programa de ejemplo *MOD386*. Con él podrá reproducir archivo MOD y VOC. Si ejecuta el programa sin parámetros, llegará a un menú de selección en el que puede elegir un archivo de sonido del directorio actual. Si especifica el parámetro *-r*, entrará al modo *repeat*. Ahora podrá elegir otro archivo, después de haber reproducido uno, hasta que pulse [ESC].

El programa emplea la Unit *Design*. Esta Unit contiene las funciones comunes para la técnica de ventana y selección de archivos. En este lugar no hablaremos de ella, ya que puede encontrarla en el CD adjunto. A continuación presentaremos y explicaremos los diferentes procedimientos del programa.

procedure Caja_Scala;

Este procedimiento dibuja la pantalla del player. El procedimiento debería ser autoexplicativo.

procedure Scala;

Scala es responsable de las informaciones en tiempo de ejecución. Visualiza las barras de volumen y actualiza las informaciones sobre la posición en el tema así como los instrumentos activos actualmente. Tampoco este procedimiento requiere mayores explicaciones.

procedure Play_the_Mod(s : string);

Este es el procedimiento más interesante para usted. Reproduce el archivo MOD, y reacciona a las entradas de usuario.

Para comenzar se reinicia la tarjeta Sound Blaster. Esto es necesario ya que no sabemos en que (desordenado) estado se encuentra. A continuación se fijan algunos parámetros necesarios para la reproducción. Para *Samfreq* puede emplear tranquilamente un valor de 22 (kHz), a menos que disponga de un 386SX-16. Mediante *carga_archivomod* se carga el archivo MOD en la memoria. Si hubo problemas durante la carga, esta función devuelve un valor distinto de *0*. En ese caso el programa visualiza un mensaje de error, y se interrumpe. De lo contrario se inicia la reproducción controlada por interrupciones del archivo MOD a través de *periodic_on*. Ahora es el momento de construir la pantalla. En un bucle se comprueba si se ha pulsado [ESC] o alguna otra tecla definida. En ese caso se reacciona adecuadamente. Esta parte del procedimiento debería ser entendida sin mayores explicaciones.

Si se pulsó la tecla [ESC] se termina el archivo MOD. Para ello se coloca en *TRUE* la variable *outfading*, lo que provoca que se realice un fundido a cero de la pieza. No se escuchará ningún sonido cuando outvolume haya alcanzado un valor menor que *1*. Hasta entonces se ha de mantener la actualización de la pantalla mediante el procedimiento *Scala*. Cuando se alcance este valor se coloca de nuevo en su valor original la interrupción del temporizador mediante *periodic_off*. Con *fin_mod* se libera de nuevo la memoria ocupada por el archivo MOD. A continuación se vuelve a reiniciar la Sound Blaster, para dejarla en su estado original.

```
procedure Play_the_Mod(s : string);
var h : byte;
    error : integer;
    li : integer;
begin;
{ if not SB16Detected then} Reset_Sb16;
 mod_SetSpeed(66);
 mod_Samplefreq(Samfreq);
 dsp_rdy_sb16 := true;
 error := lade_moddatei(s,AUTO,AUTO,Samfreq);
 if error <> 0 then begin;
   clrscr;
   writeln('Error al cargar el archivo MOD ! ');
   if error = -1 then writeln('Archivo no se encontró!');
   if error = -2 then writeln('No hay memoria suficiente!');
   halt(0);
 end;
 activar_periodic;   { apaga la reproducción periódica }

 Caja_Escala;
 ch := #255;
 while not (ch=#27) and not (upcase(ch)='X')
   and not (upcase(ch)='N') do begin;
  Scala;
  if keypressed then ch := readkey;
  case ch of
    #0  : begin;
          dch := readkey;
          case dch of
            #61 : begin; { F3 }
                  if Mastervolume > 0 then dec(Mastervolume);
                  Set_Volume(Mastervolume);
                  textbackground(black);
                  textcolor(lightblue);
                  writexy(47,2,'Volume: ');
                  textcolor(lightcyan);
                  write(Mastervolume:2);
                  ch := #255;
                end;
            #62 : begin; { F4 }
```

```
            if Mastervolume < 31 then inc(Mastervolume);
            Set_Volume(Mastervolume);
            textbackground(black);
            textcolor(lightblue);
            writexy(47,2,'Volume: ');
            textcolor(lightcyan);
            write(Mastervolume:2);
            ch := #255;
          end;
        #63 : begin; { F5 }
            if Balance > 0 then dec(Balance);
            Set_Balance(Balance);
            textcolor(lightblue);
            textbackground(black);
            writexy(58,2,'Balance ');
            textcolor(14);
            writexy(66,2,'■■■■■■■■■■■■■■■');
            textcolor(4);
            writexy(78-Balance DIV 2,2,'■');
            ch := #255;
          end;
        #64 : begin; { F6 }
            if Balance < 24 then inc(Balance);
            Set_Balance(Balance);
            textcolor(lightblue);
            textbackground(black);
            writexy(58,2,'Balance ');
            textcolor(14);
            writexy(66,2,'■■■■■■■■■■■■■■■');
            textcolor(4);
            writexy(78-Balance DIV 2,2,'■');
            ch := #255;
          end;
        else begin;
          ch := #255;
         end;
       end;
     end;
   '6' : begin;
          inc(mli);
          ch := #255;
        end;
   'f' : begin;
          filter_activ := not filter_activ;
          if filter_activ then begin;
                          filter_ein;
            textcolor(lightblue);
            textbackground(black);
            writexy(36,2,'Filter');
```

```
          textcolor(lightcyan);
          write(' ON ');
        end else begin;
                        filter_mid;
      textcolor(lightblue);
      textbackground(black);
      writexy(36,2,'Filter');
      textcolor(lightcyan);
      write(' OFF');
    end;
    ch := #255;
  end;
'4' : begin;
    if mli > 0 then
      dec(mli)
    else begin;
      if mlj > 0 then begin;
    dec(mlj);
    mli := 63
      end else begin;
    mli := 0;
    mlj := 0;
      end;
    end;
    ch := #255;
  end;
'3' : begin;
    mli := 0;
    inc(mlj);
    ch := #255;
  end;
'1' : begin;
    if mlj > 0 then begin;
      dec(mlj);
      mli := 0;
    end;
    ch := #255;
  end;
'N',
'n' : begin;
    next_song := 1;
  end;
'x' : begin;
    next_song := 255;
  end;
#27 : begin;
    next_song := 255;
  end;
else begin;
```

```
        ch := #255;
      end;
     end;
   end;
   outfading := true;
   while outvolume > 1 do begin;
     Scala;
   end;
     desactivar_periodic;
   ende_mod;
   Reset_Sb16;
 end;
```

El siguiente procedimiento interesante el *play_sound*. A él se le pasan el nombre del archivo a reproducir. Si el archivo es del tipo MOD, se llamará al procedimiento *Play_the_mod*. De lo contrario debería tratarse de un archivo VOC. Para reproducirlo se ha de reiniciar la tarjeta Sound Blaster. A continuación se visualiza un breve texto de información y se comienza con la reproducción del archivo VOC mediante *Init_Voc*. Ahora sólo ha de esperar que la variable *Voc_ready* tome el valor *TRUE* o que se pulse una tecla. Mediante *voc_pause* se puede interrumpir la reproducción del archivo VOC y mediante *voc_continue* se continúa. Cuando haya finalizado la salida (el archivo ha llegado al final) o el usuario la haya interrumpido, se ha de llamar al procedimiento *Voc_done*. Este finaliza la salida y cierra el archivo abierto. Es absolutamente necesario llamar a este procedimiento.

```
procedure write_vocmessage;
begin;
  clrscr;
  writexy(10,08,'Atención! El VOC se repetir  sin piedad !!!');
  writexy(10,10,'Terminar   con   Q  ');
  writexy(10,11,'Pausa      con   P  ');
  writexy(10,12,'Continuar  con   C  ');
  writexy(10,13,'Reiniciar  con   N  ');
  writexy(10,21,'                          E N J O Y');
end;

procedure play_sound(datname : string);
var li : integer;
    ch : char;
begin;
  for li := 1 to length(datname) do
     datname[li] := upcase(Datname[li]);
  if pos('.MOD',datname) <> 0 then begin;
    Play_The_Mod(datname);
    exit;
  end;
```

```
if pos('.VOC',datname) <> 0 then begin;
repeat
  Reset_Sb16;
  write_vocmessage;
  Init_Voc(datname);
  ch := #0;
  repeat
    if keypressed then ch := readkey;
    if ch = 'p' then begin;
      voc_pause;
      repeat
        ch := readkey;
      until ch = 'c';
      voc_continue;
    end;
  until VOC_READY or (ch = 'n') or (upcase(ch) = 'Q');
  VOC_DONE;
until upcase(ch) = 'Q';
end;
end;
```

El código del programa principal es trivial. Procure llamar primero al procedimiento *Init_the_mod*, antes de intentar reproducir sonido con cualquier rutina. Si el procedimiento no se llamó, la tarjeta Sound Blaster no está inicializada correctamente, lo que muy probablemente llevará al "cuelgue" del ordenador.

```
begin;
 Samfreq := 22;
 clrscr;
 test_systemspeed;
 cursor_off;
 interprete_commandline;
 if (Nummods = 0) and not repeatmode then begin;
textcolor(15);
textbackground(1);
clrscr;
Nummods := 1;
modd[1]:=select_archivo('*.?o?','*.?o?','','Selecciona archivo');
if modd[1] = 'xxxx' then begin;
     clrscr;
        writeln('Algún archivo MOD debería tener!');
     Cursor_on;
     halt(0);
   end;
 end;
 for i := 1 to Nummods do begin;
   if pos('.',modd[i]) = 0 then modd[i] := modd[i]+'.mod';
 end;
 Init_The_Mod;
```

```
    stereo := false;
    next_song := random(Nummods)+1;
    textcolor(lightgray);
    textbackground(black);
    write_sbconfig;
    writeln;
    writeln;
    write('                    ENTER  para seguir ...');
    readln;
    repeat
      if repeatmode then begin;
    textcolor(15);
    textbackground(1);
    clrscr;
    modd[1] := select_archivo('*.?o?','*.?o?','','');
    if modd[1] = 'xxxx' then next_song := 255
    else Play_Sound(modd[1]);
      end else
        Play_Sound(modd[next_song]);
      if next_song <> 255 then next_song := random(Nummods)+1;
    until next_song = 255;
    cursor_on;
    textmode(3);
  end.
```

Bien, eso ha sido todo lo que debe saber sobre programación de la Sound Blaster. Les deseamos mucha diversión durante sus pruebas y mejoras de los ejemplos presentados. Seguramente ofrecen los suficientes puntos de entrada como para intentar una optimización o la integración en programas propios. A lo mejor puede utilizar la Unit para su próximo juego o demo. O bien puede escribir un editor de MOD. O bien, o bien, o bien....

15.3 Así se programa un MOD-Player para la Gravis Ultrasound

La tarjeta Gravis Ultrasound es muy adecuada para reproducir archivos MOD. Ofrece la posibilidad de cargar todos los samples del módulo en la RAM de la GUS (Gravis UltraSound) y reproducir las voces directamente en correspondencia al arreglo que hay en el MOD. Los datos a reproducir ya no se han de mezclar a mano, todo esto la hace la Gravis Ultrasound por nosotros.

¿Y cómo podemos emplear estas características? Necesitamos una Unit de control, con la que podamos reproducir archivos MOD con facilidad. Todo esto lo puede encontrar en la Unit *gus_mod*. Dispone de una cuantas funciones elementales mediante las cuales puede inicializar la tarjeta, cargar un MOD y reproducirlo. Si tiene prisa, basta con que se lea el siguiente apartado sobre la sección de *Interface* de la Unit. Allí explicaremos cómo se realizan las diferentes funciones.

Estructura del MOD-Player

La reproducción de un archivo MOD es sencillo. Primero ha de inicializar la Gravis Ultrasound. Para ello necesita el port base de la GUS. Este se puede obtener mediante la función *_gus_init_env*. Devuelve el valor *TRUE*, si se pudo averiguar el port mediante la variable de entorno *Ultrasnd*, de lo contrario obtendrá el valor *FALSE*. Ahora puede inicializar la tarjeta mediante *_gus_inicializar*.

Ahora la tarjeta está lista para recibir datos. Estos se transmiten mediante la función *_gus_modload*. A la función se le ha de pasar el nombre del archivo que se quiere cargar. Devuelve el valor *TRUE* si el archivo pudo ser cargado, de lo contrario devuelve *FALSE*. Después de haber cargado el MOD en la memoria de esta forma, puede iniciar la reproducción mediante *_gus_iniciarmod*. El archivo MOD se repite automáticamente cuando llegue al final. Para terminar la salida, llame al procedimiento *_gus_finalizar_mod*. Detiene la reproducción del MOD y lo elimina de la memoria. Ahora ya sabe todo lo que necesita para integrar en sus programas un MOD-Player para la Gravis Ultrasound.

Variables importantes del GUS-MOD-Player

Si desea realizar por ejemplo un MOD-Player, no sólo necesita la posibilidad de reproducir un MOD. Sino que también querrá poder influir sobre el programa en tiempo de ejecución y poder darle al usuario informaciones sobre el estado actual de player. Todo esto se puede regular mediante las variables de la Unit.

Play_Chanel : array[1..14] of byte;

Los valores de las entradas de *Play_Chanel* tienen bien el valor *1* o bien el valor *0*. *1* significa que se está reproduciendo el canal correspondiente y *0* apaga el canal. De esta forma se puede obtener un *Muting* de los canales con sencillez.

Canales : array[0..31] of PCanalInfo;

En este array se guardan las informaciones de los distintos canales. El array se compone de punteros que apuntan a las diferentes estructuras de canal. La estructura de un canal tiene el siguiente aspecto:

```
TCanalInfo = record
  InstNr     : byte;        { Variables referidas al hardware }
  Mempos     : longint;
  Fin        : longint;
```

```
    Inicio_bucle : longint;
    Fin_bucle    : longint;

    Volume       : integer;          { Variables referidas al MOD }
    Frecuencia   : word;
    Looping      : byte;
    Tono         : integer;
    Tono_Inicio: integer;
    Tono_Final   : integer;
    Efecto       : byte;
    Operando     : byte;

    Efectox,                         { Variables referidas a efecto }
    Efectoy      : integer;
    Appegpos     : integer;
    slidespeed   : integer;
    vslide       : integer;
    retrig_count : byte;
    vibpos       : byte;
    vibx         : byte;
    viby         : byte;
  end;
```

Las diferentes entradas tienen las siguientes funciones:

InstNr *InstNr* contiene el número del instrumento a reproducir, tal y como se emplea en el archivo MOD. Puede contener un valor entre *1* y *31*.

Mempos

En *Mempos* puede encontrar la posición inicial de la voz activa en la RAM de la Gravis Ultrasound.

Fin Fin designa la posición final de la voz en la GUS-RAM.

Inicio_bucle

Las voces puede repetirse. En ese caso podrá encontrar aquí la posición inicial del looping en la RAM de la GUS.

Fin_bucle

El final correspondiente del bucle se encuentra en esta variable. También ella se refiere a la RAM de la Gravis Ultrasound.

Volume Aquí se encuentra el volumen con el que se reproduce la voz. Se emplean los volúmenes del archivo MOD, que pueden estar entre *0* y *63*.

Frecuencia

En este lugar se almacena la frecuencia de la voz, tal y como está almacenada en el archivo MOD. Esta no es la verdadera frecuencia de reproduc-

ción, sino la que se emplea en el archivo MOD, que aún ha de ser convertida. Se necesita sobre todo para los diferentes efectos.

Looping

Esta variable indica si la voz se repite o no. Contiene *0* si la voz no se repite y *8* de lo contrario. Esta cifra se necesita para la programación de la Gravis Ultrasound. Si el tercer bit de una voz está activado, esto significa que se repite.

Tono, Tono_inicio, Tono_final

Estas variables se necesitan para el cálculo en un MOD. Lamentablemente no es suficiente con una sola variable, ya que frecuentemente se han de guardar los valores de forma temporal.

Efecto Aquí se guarda el efecto actual que se encuentra en el archivo MOD.

Operando

Aquí se encuentra el operando correspondiente al efecto.

Efectox, Efectoy

En *Efectox* se encuentran los 4 bits superiores del operando y en *Efectoy* los 4 bits inferiores.

Appegpos

En un *Arpeggio* siempre se tocan tres notas distintas en secuencia. Esta variable indica cuál de estos tonos está activo actualmente.

slidespeed

Esta variable se necesita para el *Frecuence Sliding*. Indica la velocidad con la que se realiza el desplazamiento.

vslide Análogamente a *slidespeed* esta variable indica la velocidad de un Volume-Sliding.

retrig_count

Esta variable se necesita para la realización de un efecto de retriggering. Indica la posición del nuevo inicio de la voz.

vibpos Aquí puede encontrar la posición actual en la tabla de vibratos.

vibx,viby

Los parámetros x e y del efecto de Vibrato se almacenan en estas dos variables.

Instrumentos : array[0..31] of PInstrumentnrInfo

Las informaciones para los 31 instrumentos de un archivo MOD se pueden encontrar aquí. La variable es un array con punteros a la siguiente estructura:

```
PInstrumenrInfo = ^TInstrumentInfo;
TInstrumentInfo = record
  Nombre       : string[22];
  Mempos       : longint;
  Fin          : longint;
  l_Inicio     : longint;
  l_Fin        : longint;
  Tamanyo      : word;
  Inicio_bucle: word;
  Fin_bucle    : word;
  Volume       : word;
  Looping      : byte;
end;
```

Las diferentes entradas tienen el siguiente significado:

Name Aquí se encuentra el nombre de cada uno de los instrumentos.

Mempos

En *Mempos* se guarda la posición de los datos de sampling en la RAM de la GUS.

Fin *Fin* designa la posición final del instrumento en la RAM de la Gravis Ultrasound.

L_Inicio

Los instrumentos puede repetirse. En este caso se encuentra aquí la posición inicial del looping en la RAM de la GUS.

L_Fin El final correspondiente del looping se puede encontrar en esta variable. También ella se refiere a la RAM de la GUS.

Tamanyo

Aquí se guarda la longitud en bytes de la voz guardada en el archivo MOD.

Inicio_bucle

Inicio_bucle contiene el offset del instrumento en el que comienza el looping.

Fin_bucle

Aquí puede encontrar el offset en los datos de sampling del instrumento, en el que finaliza el looping.

Volume El volumen de la voz se almacena aquí. Puede estar entre 0 y 63.

Looping

Esta variable indica su la voz se repite o no. Contiene *0* si la voz no se repite, de lo contrario vale *8*. Este valor se necesita para la programación de la GUS.

MOD_Voces : word

Según la cantidad de voces que tenga el tema, aquí puede encontrar el valor *4* u *8*.

Mod_Patternsize : word

En esta variable se guarda el tamaño del patrón. Se obtiene según la fórmula *Voces_MOD * 256*. De manera que un patrón de cuatro voces mide 1024 bytes.

Stop_TheVoice : array[1..8] of boolean;

Este array sirve para detener una voz determinada. Si coloca la variable correspondiente a la voz en *TRUE*, se detiene la voz.

Modinf : Modinfo;

Modinfo contiene informaciones globales sobre el MOD.

Titulo

Aquí se guarda el título del tema.

Patt_num

Aquí puede encontrar el número de patrones definidos.

Runinf : Runinfo

Aquí están las informaciones de tiempo de ejecución, que necesita por ejemplo para la realización un MOD-Player propio o la sincronización de gráficos con el MOD. Se encuentran definidas las siguientes entradas:

Caida[1..8]

Aquí se simula un pseudoecualizador para cada voz. El valor siempre se carga con *63* cuando se toca una nota. En cada tick se decrementa este valor.

Linea

Linea designa la línea del patrón que se está ejecutando. Puede estar entre *1* y *63*.

Pattnr

En esta variable se encuentra la posición con respecto el arreglo. Mediante ella y la variable *Linea* se pueden sincronizar muy bien determinados eventos o animaciones con la música.

Volumes [1..8]

Representa al volumen actual de cada voz.

Speed, BMP

Aquí se puede leer la velocidad del MOD. *Speed* tiene un valor entre *1* y *15* e indica la velocidad en relación a la velocidad base actual en BPM (por defecto: 125).

chpos : array[1..8] of byte

Aquí se puede introducir la posición de las voces. Son posibles valores desde *0* para la izquierda, hasta *15* para la derecha.

VibratoTable : array[0..63] of integer;

En *VibratoTable* se guardan todas las entradas necesarias para el Vibrato. La tabla describe, con sus 64 entradas, exactamente una oscilación, es decir, los valores tienen sus extremos en las posiciones *15* y *47*, valiendo *0* para *0* y *32*. Esta tabla se necesita tanto para el Vibrato como para el efecto de Tremolo.

Rutinas centrales del MOD-Player

Vamos a por lo realmente interesante de la Unit, los procedimientos. ¿Qué es lo que necesitamos para un MOD-Player? Bien, por una parte están los procedimientos con los que cargamos el MOD del disco a la memoria. Después hemos de llevar los datos de sampling de alguna manera hasta la GUS. Y también necesitamos rutinas para reproducir el MOD.

La carga de un MOD

Así que veamos primero cómo se carga un MOD. En esta Unit esto lo realiza la función *_gus_modload*. Se le ha de pasar el nombre completo del archivo MOD a cargar. Si se pudo realizar esta operación con éxito, la función devuelve el valor *TRUE*. Sino, se interrumpe con el resultado *FALSE*.

Primero la función reserva la memoria para las estructura de canal y las informaciones de instrumento. A continuación se pone a *0* la línea en *Runinf* y Pattnr a *-1*. Con ello se inicia la rutina de reproducción al principio del MOD.

Ahora se puede abrir el archivo. El procedimiento guarda la pantalla y visualiza un gráfico ANSI que indica la carga del archivo MOD (*Save_Screen; display_loading*). Estas dos llamadas pueden omitirse tranquilamente en su player, o bien las puede sustituir por sus propios procedimientos de pantalla.

LongRest se carga ahora con el tamaño de archivo *-1084*. En la variable se encuentra el tamaño del archivo MOD sin la cabecera. Mediante ella se determinan el número de patrones en el MOD. A continuación se comprueba con ayuda de la identificación si se trata de un archivo MOD y cuántas voces tiene. En correspondencia se fijan las variables *Voces_Mod* y *MOD_Patternsize*. Ahora se puede leer la cabecera completa. Comenzamos con el nombre del archivo MOD. Este es una cadena terminada en cero, y no se puede procesar directamente en Pascal. Para convertirla al formato de Pascal empleamos la función *ConvertString*. A ella se le han de pasar un puntero a una zona de memoria en la que se encuentra la cadena, y su longitud máxima. La función devuelve una cadena Pascal, que podemos emplear en nuestro programa.

Ahora nos queda obtener la longitud del tema, es decir, cuántas entradas hay definidas finalmente en los arreglos. Esta longitud de tema no tiene nada que ver con la cantidad de patrones almacenados en el MOD. A continuación se ha de leer el arreglo de 128 bytes de largo, antes de poder cargar los instrumentos.

El problema que se nos plantea es que hay MOD con 31 voces, pero que algunos módulos antiguos sólo tienen 15 voces. Sería algo complicado (aunque practicable), cargar las voces en dos segmentos de código distintos. Aquí sin embargo preferimos emplear dos variables. Por una parte la variable *vh.Num_Inst*. Contiene el número real de instrumentos que se encuentran en el archivo MOD. Por otra tomamos *pdi* (de Posición Dependiente de Instrumentos). En el caso de 31 instrumentos tiene el valor 0, para 15 instrumentos el valor *-16 * 30*. Este valor tiene la siguiente explicación: Hay 16 instrumentos menos, y cada información de instrumento mide 30 bytes. Internamente trabajamos con los offsets de un archivo de 31 voces, de modo que he:nos de restar el offset de los 16 instrumentos.

Después de haber conseguido estas dos variables, podemos definir en un bucle todos los instrumentos definidos. Esto se realiza de la siguiente manera:

Primero leemos el nombre de instrumento de 22 bytes. También aquí se trata de una cadena terminada en *0*, que hemos de convertir en una cadena de Pascal mediante la función *ConvertString*. Después se lee el tamaño de los samples en bytes. Por desgracia, el tamaño leído aún no es el real, sino que es una palabra de Amiga. Esto significa que hemos de permutar los bytes alto y bajo, para pasarlo al formato de PC. Después nos queda multiplicar el valor obtenido por dos, ya que se trata de un valor Word, y nosotros necesitamos la longitud en bytes.

Este sistema de conversión se aplica a todos los valores en palabras de Amiga. El valor del volumen que viene a continuación sin embargo es un byte, y se encuentra entre *0* y *63*. No es necesario convertirlo. Ahora vienen la posición del looping del sample (referido como offset en el sample) y la longitud de este looping. Ambos son datos en palabras de Amiga, y por ello se han de convertir. De la longitud podemos determinar directamente la posición final del sample, para que no hayamos de calcular innecesariamente durante el procesamiento del MOD.

La función de carga comprueba a continuación si el instrumento correspondiente se repite, es decir, si la diferencia entre el inicio del looping y el final del mismo es >= que 10. En este caso se le asigna el valor *con_loop* (=*8*). De lo contrario se le asigna el valor *no_loop*. Finalmente se decrementa *LongRest* en el tamaño del MOD. Con ello esta variable contiene un valor después de leer todas las informaciones de instrumento, que corresponde al número de bytes de todos los patrones definidos. Así se puede determinar el número de patrones existentes mediante una división del valor entre el tamaño de los diferentes patrones. Y estos se leen ahora en un bucle.

Finalmente se realiza la carga de los instrumentos. También estos se cargan en un bucle. Para ello se emplea el procedimiento *Carga_instrumento*, que luego veremos en detalle. En el bucle se realiza una manipulación de la memoria de pantalla después de la llamada del procedimiento, memoria que se direcciona mediante la variable *Screen*. Aquí siempre se modifica el atributo (en este caso el color de texto) del carácter en la barra de progresión del "Ansi de carga". Esta es una característica que sólo se necesita para el MOD-Player TCP. Para sus rutinas lo puede omitir, o sustituirlo por sus propios gráficos.

Ahora ya ha cargado el archivo MOD completo en la memoria, así que lo puede cerrar. En la función de carga se inicializan ahora algunas variables importantes. Primero se determinan correctamente todas las voces de la Gravis Ultrasound. Después se ordenan los canales en una especie de semicírculo. Esto quiere simular

un sonido espacial. Para ello se le asignan valor entre *0* y *15* a las variables *chpos[1]* hasta *chpos[Voces_Mod]*. Mediante *_gus_set_chanelpos* se le asignan estas variables a las diferentes voces. Finalmente se ajusta la velocidad de interrupciones al valor estándar 125 y se restaura la pantalla mediante el procedimiento *restore_screen*. En su propia rutina de carga puede prescindir de la restauración de la pantalla, ya que sólo se necesita para TCP.

```
function _gus_cargamod(name : string) : boolean;
{
 Carga el archivo MOD pasado en Nombre. Presupone que el archivo in-
dicado
 existe en la vía de acceso, y no está protegida contra escritura.
}
var dummya : array[0..30] of byte;{ para tratamiento de cadenas      }
    daptr : pointer;              { puntero a dummya                 }
    dumw : word;                  { variable dummy para lectura      }
    dumb : byte;
    LongRest : longint;           { para obtener el nº de patrones   }
    li : integer;
    Identificador : array[1..4] of char;{ identificación archivo MOD}
    ias : integer;                { pos. inicial según instrumentos  }
begin;
  U_Ram_freepos := 32;
  for li := 0 to 15 do begin;
    new(Canales[li]);
    Canales[li]^.vibpos := 0;
  end;
  for li := 0 to 31 do begin;
    new(Instrumentos[li]);
    Instrumentos[li]^.Nombre := '';
  end;

  runinf.Linea := 0;
  runinf.Pattnr := -1;
  tickcounter := 0;
  ticklimit := 6;
  runinf.speed := 6;
  runinf.bpm := 125;
  ias := 0;
  daptr := @dummya;

  assign(gusmf,name);             { abrir archivo + inicializar long}
  reset(gusmf,1);

  save_screen;
  display_loading(name);
  LongRest := filesize(gusmf);
  LongRest := LongRest - 1084;
```

```
seek(gusmf,1080);                { comprobar si MOD con 15/31 voces}
Blockread(gusmf,Identificador,4);
if pos(Identificador,ModIdentificadoren) = 0 then begin;
  { 15 Voces ? }
  seek(gusmf,600);
  Blockread(gusmf,Identificador,4);
  if pos(Identificador,ModIdentificadoren) = 0 then begin;
    { Keine gültige .MOD-Datei }
    writeln('No es archivo .MOD válido!!!');
    halt(0);
  end else begin;
    Instancia_MOD := 15;
    ias := -16*30;
  end;
end;

if (Identificador = MODId[1]) or     {¿nº voces del archivo MOD? }
   (Identificador = MODId[2]) or
   (Identificador = MODId[3])
then begin;
  MOD_Voces := 4;
  Tamanyo_Patron_MOD := 4*256;
end else
  if (Identificador = CHn6) then begin;
    _gus_cargamod := false;
    exit;
  end else
    if (Identificador = CHn8) then begin;
 MOD_Voces := 8;
 Tamanyo_Patron_MOD := 8*256;
    end;

seek(gusmf,0);
Blockread(gusmf,dummya,20);      { averiguar nombre del archivo  }
vh.SongNombre := ConvertString(daptr,20);
seek(gusmf,950+ias);             { longitud de tema en patrones  }
Blockread(gusmf,vh.LongTema,1);
seek(gusmf,952+ias);             { leer arreglos                 }
Blockread(gusmf,vh.Arrang,128);

vh.Num_Inst := Instancia_MOD;       { leer instrumentos (15/31)  }
seek(gusmf,20+ias);

for li := 1 to 32 do Instrumentos[li]^.Nombre := '';

for li := 1 to vh.Num_Inst do begin;
  Blockread(gusmf,dummya,22);    { nombre de instrumentos        }
  Instrumentos[li]^.Nombre := ConvertString(daptr,22);
```

```
  Blockread(gusmf,dumw,2);          { longitud del sampling          }
  Instrumentos[li]^.Tamanyo := swap(dumw) * 2;

  Blockread(gusmf,dumb,1);          { leer volumen                   }
  Blockread(gusmf,dumb,1);
  Instrumentos[li]^.Volume := dumb;

  Blockread(gusmf,dumw,2);          { leer inicio del bucle          }
  Instrumentos[li]^.Inicio_Bucle := swap(dumw) * 2;
  Blockread(gusmf,dumw,2);

  dumw := swap(dumw) * 2;           { leer fin de bucle de Start+Länge}
  Instrumentos[li]^.Loop_Fin:=Instrumentos[li]^.Inicio_Bucle+dumw;

  if (Instrumentos[li]^.Loop_Fin -        { ¿Looping en instrumento?}
 Instrumentos[li]^.Inicio_Bucle) >= 10 then begin;
    Instrumentos[li]^.Looping := mit_loop;
  end else begin;
    Instrumentos[li]^.Looping := no_loop;
  end;
  Dec(LongRest,Instrumentos[li]^.Tamanyo);
end;

Vh.Num_Patts := LongRest DIV Tamanyo_Patron_MOD ; { ¿nº patrones? }
seek(gusmf,1084+ias);

for li := 1 to Vh.Num_Patts do begin; { leer patrones              }
  dos_getmem(Pattern[li],Tamanyo_Patron_MOD );
  Blockread(gusmf,Pattern[li]^,Tamanyo_Patron_MOD );
end;

for li := 1 to vh.Num_Inst do begin; { leer instrumentos           }
  Carga_Instrumento(li);
  screen[16,23+li].a := 5;
end;

close(gusmf);

for i := 1 to 31 do begin; { inicializar variables de canal        }
  u_VoiceBalance (i,7) ;
  u_VoiceVolume (i,0) ;
  u_VoiceFreq (i,12000);
  U_StartVoice(i,Stop_Voice);
  u_Voicedata(0,0,0,i);
end;
runinf.Linea := 0;                    {inicializar variables ejec.}
runinf.Pattnr := -1;
tickcounter := 0;
```

```
ticklimit := 6;
runinf.speed := 6;
runinf.bpm := 125;
if MOD_Voces = 4 then begin;  { ordenar canales en semicírculo    }
  chpos[1] := 2;
  chpos[2] := 5;
  chpos[3] := 9;
  chpos[4] := 12;
  _gus_set_chanelpos;;
end;
if MOD_Voces = 8 then begin;
  chpos[1] := 1;
  chpos[2] := 3;
  chpos[3] := 5;
  chpos[4] := 7;
  chpos[5] := 7;
  chpos[6] := 9;
  chpos[7] := 11;
  chpos[8] := 13;
  _gus_set_chanelpos;;
end;

Nueva_Vel_Interrup(runinf.bpm);
restore_screen;
Modinf.Voces   := MOD_Voces;  { MOD-Infos constantes en estruct.  }
Modinf.Titulo    := vh.Songname;  { para pasar                    }
Modinf.Num_Pat := Vh.LongTema;
_gus_cargamod     := true;
end;
```

Veamos ahora el procedimiento de carga de los instrumentos. Tiene el nombre *carga_instrumento* y se le ha de pasar el número del instrumento a cargar. Si es mayor de 10 bytes (¡si no de todas formas sólo habrá basura ahí dentro!), se carga en la RAM de la Gravis Ultrasound. Para ello el procedimiento primero lee el sample en un buffer de 64 KBytes. Después se busca la siguiente dirección de párrafo (dirección divisible entre 16) de la GUS-RAM. Esta posición es la posición inicial del sample. Se le asigna a la variable de instrumento *Mempos*. Ahora se pasa el sample a la RAM de la Gravis Ultrasound mediante el procedimiento *Ultra_Mem2Gus*. A continuación se vuelve a liberar el buffer reservado. Después se escriben la posición inicial del looping y el final de la voz en las informaciones de instrumento de la GUS-RAM. Veamos cómo se realizó la programación de todo ello:

```
procedure Carga_Instrumento(Nr : byte);
{
 Carga el instrumento con el nº nr a la RAM-GUS
}
```

```
var gr : longint;
    samp : pointer;
begin;
   gr := Instrumentos[nr]^.Tamanyo;
   if gr > 10 then begin;        { sólo cargar si > 10, sino eh crap !}
     dos_getmem(samp,gr);
     Blockread(gusmf,samp^,gr);
     U_Ram_freepos := U_Ram_freepos + (16-(U_Ram_freepos MOD 16));
     Instrumentos[nr]^.Mempos := U_Ram_freepos;
     Ultra_Mem2Gus(samp,Instrumentos[nr]^.Mempos,gr);
      dos_freemem(samp);            { inicial. variables de voces  }
     Instrumentos[nr]^.l_start :=
    Instrumentos[nr]^.Mempos + Instrumentos[nr]^.Inicio_Bucle;
     if Instrumentos[nr]^.Looping = Mit_loop then begin;
     Instrumentos[nr]^.ende :=
    Instrumentos[nr]^.Mempos + Instrumentos[nr]^.Loop_Fin;
     end else begin;
       Instrumentos[nr]^.ende := Instrumentos[nr]^.Mempos + gr - 25;
     end;
     Inc(U_Ram_Freepos,gr);        { mover puntero de gestión }
   end;
end;
```

La reproducción de un MOD cargado

Después de haber cargado el MOD, naturalmente querremos reproducirlo. Esto plantea una cuestión. ¿Cual es manera más elegante de realizar esto? Teóricamente se podría incrementar una variable en un bucle, o sincronizar con el retrazado vertical de la tarjeta gráfica. Pero esto es demasiado impreciso por una parte y lo que es peor, extremadamente poco flexible. Así que preferimos emplear la interrupción del temporizador. Para ello hemos de reprogramar la interrupción con una nueva frecuencia y desviar la rutina del Handling a nuestro propio procedimiento. En éste se ha de incrementar un contador con cada tick (es decir, cada llamada). Cuando este contador alcance la velocidad *Speed* ajustada en el archivo MOD, el MOD se ha de avanzar una línea. De lo contrario se han de procesar eventuales efectos en tiempo de ejecución. ¡No olvide enviar un *EOI* (End of Interrupt) al final, mediante la escritura del valor *20h* al port *20h*!

Nuestro procedimiento para ejecutar el MOD tiene el nombre *_gus_iniciarmod*. Calcula la frecuencia a ajustar para el temporizador de los BPM (Beats per Minute) activos en el MOD. Éste último se programa a través de los ports *43h* y *40h*. El temporizador se ajusta de forma que dispare una interrupción periódica. Antes de asignarle nuestro propio procedimiento al temporizador, hemos de guardar la interrupción antigua. Esto es necesario para que después los podamos restaurar. Ya que si terminamos nuestro programa, y el temporizador sigue apuntando a nuestro procedimiento, esto con toda seguridad colgará el sistema.

De la restauración de la interrupción se encarga el procedimiento *restaurar_timer*. Restaura el temporizador a su frecuencia original, y devuelve la interrupción a su rutina habitual.

Mediante la utilización de la interrupción del temporizador se puede realizar muy fácilmente una función de pausa. Si quiere interrumpir la reproducción, simplemente desvíe el temporizador a otra rutina que no haga otra cosa que enviar *EOI*. Antes sin embargo debería colocar el volumen de los diferentes canales a *0*. Cuando quiera continuar con la reproducción, simplemente vuelva a colocar la interrupción a la rutina del player y encienda de nuevo las voces. Veamos cómo se resolvió esto en la Unit:

```
procedure mytimer; interrupt;
{
 Mi interrupción de temporizador
}
begin;
  tick_effects;
  inc(tickcounter);
  if tickcounter >= ticklimit then begin;
    Tickcounter := 0;
    Play_Pattern_gus;
  end;
  Port[$20] := $20;
end;

procedure NoHacerNada; interrupt;
{
 interrupción dummy. Se pasa a ella, cuando se para la reproducción.
}
begin;
  port[$20] := $20;
end;

procedure _gus_iniciamod;
{
 inicia la reproducción del archivo MOD mediante la interrupción del
 temporizador. ¡El archivo MOD ha de estar cargado!
}
var Contador : word;
    loz,hiz : byte;
begin;
 Contador := 1193180 DIV interrupt_speed;
 loz := lo(Contador);
 hiz := hi(Contador);
 asm
```

```
    cli
    mov dx,43h
    mov al,36h
    out dx,al
    mov dx,40h
    mov al,loz
    out dx,al
    mov al,hiz
    out dx,al
   end;
   getintvec(8,TimerAntiguo);
   setintvec(8,@Mytimer);
   asm sti end;
  end;


  procedure _gus_player_pause;
  {
   detiene la reproducción mediante la interrupción de temporizador
  }
  var li : integer;
  begin;
   setintvec(8,@Tue_nichts);
   for li := 0 to 31 do
     u_VoiceVolume (li,0) ;
  end;

  procedure _gus_player_continue;
  {
   continúa la reproducción mediante la interr. de temporizador
  }
  var li : integer;
  begin;
   setintvec(8,@Mytimer);
   for li := 1 to 31 do
     Voice_Rampin(li,Canales[li]^.volume);
  end;

  procedure Reponer_IntTimer;
  {
   repone la interr. de temporizador
  }
  begin;
   asm
    cli
    mov dx,43h
    mov al,36h
    out dx,al
    xor ax,ax
```

```
    mov dx,40h
    out dx,al
    out dx,al
  end;
  setintvec(8,TimerAntiguo);
  asm sti end;
end;
```

Cada vez que el MOD avanza una línea, se llama el procedimiento *play_pattern_gus*. Es responsable de la reproducción de las voces en función de los arreglos y los patrones. Primero avanza una línea en el MOD. Ahora se comprueba si el número de línea incrementado es mayor de *64*. En ese caso el MOD ha de avanzar un patrón, y la línea se coloca al principio, es decir a *1*. Si el número de siguiente patrón, es decir, la posición en el arreglo, sobrepasa la longitud del tema, se vuelve a colocar al principio del mismo. Después se para la línea actual del patrón en el array *La_Linea*. De esta forma, el acceso a los diferentes bytes es algo más sencillo.

A continuación se procesa esta línea. Primero se comprueba para todas las voces si han de ser reproducidas. Este es el caso cuando la entrada correspondiente contiene el valor *1* en *play_chanel*. Ahora se toma de la línea el tono a reproducir, el instrumento a emplear y el efecto correspondiente. Después se ha de distinguir si el tono está activado, y si se ha de emplear otro instrumento.

Si la entrada para el tono no es *0*, se indicó una frecuencia. Ahora se ha de comprobar primero si el efecto que se encuentra en la línea es *3*. En ese caso el tono representa un tono de destino, que ha de ser alcanzado. De lo contrario el tono es para ser reproducido directamente. En función del tipo, el tono es asignado a la variable de canal *Tono_Final* o *Tono* y *Tono_inicio*.

Si la entrada para el instrumento es distinta de cero, se han de carga de nuevo las variables para los parámetros de instrumento. Primero se transmite el número del instrumento a reproducir, a continuación las diferentes posiciones de relevancia en la RAM de la Gravis Ultrasound. Ahora sólo falta fijar el volumen y transmitir un posible looping. Es importante escribir los parámetros obtenidos directamente en la voz correspondiente, ya que manejo de la nota está orientado a tonos.

Después de haber procesado los instrumentos, así como la activación de la nota, debe dedicarse al tratamiento de efectos. Estos los regulamos mediante una procedimiento propio, que le presentaremos a continuación. Simplemente es demasiado complejo como para colocarlo aquí. Esto sólo serviría para complicar el código innecesariamente.

Finalmente ha de comprobar si se ha de reproducir un tono (*Tono* <> *0*). Para ello se ha de averiguar de nuevo la frecuencia del canal. Esta se puede obtener mediante una división de la frecuencia base para el número de voces entre la que se especifica en al archivo MOD. Ahora podrá fijar esta frecuencia mediante el procedimiento *u_VoiceFreq*. Si ha especificado el comando *Set_Volume*, ha de comprobar si se encuentra en los límites permitidos. Fije entonces el volumen del instrumento y detenga la reproducción de la voz antigua. Ahora se puede iniciar la voz con los nuevos datos programados. Esta es la conversión al Pascal:

```
procedure play_pattern_gus;
{
  este procedimiento se llama periódicamente. Reproduce una línea del
  archivo MOD
}
var li      : integer;
    dumw    : word;
    L_Linea : array[1..8,0..3] of Byte;
    Efecto  : byte;
    Nota    : word;
    Inst    : byte;
begin;
{
  ##################################################################
  ###   Avanzar en el MOD                                        ###
  ##################################################################
}
  inc(runinf.Linea);                    { avanzar una línea        }
  if runinf.Linea > 64 then runinf.Linea := 1;
  if runinf.Linea = 1 then begin; { nuevo patrón?                  }
    inc(runinf.Pattnr);
    if runinf.Pattnr > vh.LongTema then runinf.Pattnr := 1;
  end;
                                        { cargar notas             }
  move(ptr(seg(pattern[vh.Arrang[runinf.Pattnr]+1]^),
       ofs(pattern[vh.Arrang[runinf.Pattnr]+1]^)+
          (runinf.Linea-1)*4*Mod_Voces)^,
       L_Linea,4*8);
{
  ##################################################################
  ###   procesar las voces                                       ###
  ##################################################################
}
  for li := 1 to MOD_Voces do begin;
    if play_chanel[li] = 1 then begin;
      stop_Thevoice[li] := false;

      Nota  := ((L_Linea[li,0] AND $0f) shl 8)+L_Linea[li,1];
```

```
Inst:=(L_Linea[li,0] AND $f0)+((L_Linea[li,2] AND $F0) SHR 4);
Canales[li]^.Efecto    := L_Linea[li,2] AND $0f;
Canales[li]^.Operando := L_Linea[li,3];

Canales[li]^.Start_Nota := oldv[li];

if Nota <> 0 then begin;  { ¿tono registrado? }
  if Canales[li]^.Efecto = 3 then begin;
    Canales[li]^.Ziel_Nota := Nota;
  end else begin;
    Canales[li]^.Nota := Nota;
    Canales[li]^.Start_Nota := Nota;
    oldv[li] := Canales[li]^.Start_Nota;
  end;
end;

 If Inst <> 0 then begin; {¿¿¿emplear nuevo instrumento??? }
 Canales[li]^.InstNr        := Inst;
 Canales[li]^.Mempos        := Instrumentos[Canales[li]^.InstNr]
 ^.Mempos;
Canales[li]^.Inicio_Bucle := Instrumentos[Canales[li]^.InstNr]
 ^.l_start;
 Canales[li]^.Fin           := Instrumentos[Canales[li]^.InstNr]
 ^.ende;
 Canales[li]^.volume        := Instrumentos[Canales[li]^.InstNr]
 ^.volume;
 Canales[li]^.Looping       := Instrumentos[Canales[li]^.InstNr]
 ^.Looping;
u_Voicedata(Canales[li]^.Mempos,Canales[li]^.Inicio_Bucle,
                 Canales[li]^.Fin,li);
end;
Canales[li]^.Retrig_count := 0;

Initialisiere_Efectoe(li);

If (Nota <> 0) then begin; { nota comenzada }
  Canales[li]^.Frecuencia := longint(Voice_Base[14] div
Canales[li]^.Start_Nota);
  u_VoiceFreq(li,Canales[li]^.Frecuencia); { fijar frecuencia}
  if Canales[li]^.Efecto = $c then begin; { Extra, si no dema
  siado temprano}
  if Canales[li]^.Operando > 63 then Canales[li]^.Operando := 63;
  if Canales[li]^.Operando < 1 then
    begin
      Canales[li]^.volume := 0;
      u_VoiceVolume(li,0);
      U_StartVoice(li,Stop_Voice);
      stop_Thevoice[li] := true;
    end else begin
      Canales[li]^.volume := Canales[li]^.Operando;
```

```
            u_VoiceVolume(li,Canales[li]^.volume);
            Runinf.Volumes[li] := 63;
          end;
        end else begin;
          if Canales[li]^.volume > 63 then Canales[li]^.volume := 63;
           voice_rampin(li,Canales[li]^.volume);
           Runinf.Volumes[li] := 63;
        end;

        U_StartVoice(li,Stop_Voice); { para voz antigua }

        u_Voicedata(Canales[li]^.Mempos,Canales[li]^.Inicio_Bucle,
            Canales[li]^.Fin,li);
        if not stop_Thevoice[li] then begin; { iniciar nueva voz }
          U_StartVoice(li,Play_Voice+Bit8+Canales[li]^.Looping+Unidirect);
          Runinf.Nivel[li] := Canales[li]^.volume * 4; {ecualizador}
        end;
      end; { nota iniciada }
    end else begin;
      u_VoiceVolume(li,0);
    end;
  end; {for}
end;
```

Lo realmente interesante es la realización de los efectos. Para ello se ha de distinguir entre el procedimiento para la inicialización de los mismos y el procedimiento que actualiza los efectos en cada tick del MOD. Observemos primero el procedimiento de inicialización, que es llamado por *play_pattern_gus*.

El procedimiento comprueba primero si el efecto no es *0*. Si es *0*, se termina el procedimiento, ya que no hay ningún efecto presente. Ahora se comprueba en un bucle *Case*, qué efecto se ha de aplicar. Si define un procedimiento propio para cada uno, puede trabajar con una Jump-Table, en la que guarda todas las direcciones de procedimientos de efecto. Ya que en el caso de un player para la Gravis Ultrasound la velocidad no es tan crítica (bueno, tampoco hay que pasarse), podemos aguantar la estructura *Case*. Al fin y al cabo tiene la ventaja de ser un poco más clara.

En esta estructura *Case* se realizan ahora las modificaciones correspondientes en los parámetros del canal, en función del número del efecto adecuado. Por simple sencillez, vamos a repasar todos los efectos incluidos:

Efecto 0 - Arpeggio

En este efecto se reproduce el tono en tres frecuencias diferentes y en rápida secuencia. La frecuencia almacenada en *Tono_inicio* es la frecuencia inicial. El nibble x es el primer valor en el cual se ha de aumentar la misma y el nibble y el segundo.

En base a esto se ha de calcular y fijar la frecuencia de nuevo en cada tick.

```
procedure Initialisiere_Efectoe(nr : byte);
var swaplong      : longint;
    vibswap       : integer;
begin;
  if Canales[nr]^.Efecto = 0 then exit;
  case Canales[nr]^.Efecto of
    0 : begin; { Appegio }
          Canales[nr]^.Appegpos := 0;
          Canales[nr]^.Efectox := Canales[nr]^.Operando shr 4;
          Canales[nr]^.Efectoy := Canales[nr]^.Operando and $0f;

          inc(Canales[nr]^.Appegpos);
          case (Canales[nr]^.Appegpos MOD 3) of
            0 : begin; {ap = 3 !}
                  Canales[nr]^.Start_Nota :=
                  Canales[nr]^.Nota + Canales[nr]^.Efectoy;
                end;
            1 : begin; {ap = 1 !}
                  Canales[nr]^.Start_Nota :=
                  Canales[nr]^.Nota;
                end;
            2 : begin; {ap = 2 !}
                  Canales[nr]^.Start_Nota :=
                  Canales[nr]^.Nota + Canales[nr]^.Efectox;
                end;
          end;
          if Canales[nr]^.Start_Nota < 1 then
            Canales[nr]^.Start_Nota := 1;
          Canales[nr]^.Frecuencia :=
            longint(Voice_Base[14] div Canales[nr]^.Start_Nota);
          u_VoiceFreq(nr,Canales[nr]^.Frecuencia);
        end;
```

Efecto 1 - Portamento up

En el caso de *Portamento up* se decrementa la frecuencia especificada en el valor indicado en el operando. La frecuencia que así se genera se ha de fijar de nuevo en cada tick.

```
    1 : begin;  { Portamento up }
          dec(Canales[nr]^.Start_Nota,Canales[nr]^.Operando);
          if Canales[nr]^.Start_Nota < 1 then
            Canales[nr]^.Start_Nota := 1;
          Canales[nr]^.Frecuencia :=
            longint(Voice_Base[14] div Canales[nr]^.Start_Nota);
          u_VoiceFreq(nr,Canales[nr]^.Frecuencia);
        end;
```

Efecto 2 - Portamento down

Este efecto se comporta como *Portamento up*, sólo que aquí se incrementa y no decrementa la frecuencia.

```
2 : begin;  { Portamento down }
      inc(Canales[nr]^.Start_Nota,Canales[nr]^.Operando);
      if Canales[nr]^.Start_Nota < 1 then
        Canales[nr]^.Start_Nota := 1;
      Canales[nr]^.Frecuencia :=
        longint(Voice_Base[14] div Canales[nr]^.Start_Nota);
      u_VoiceFreq(nr,Canales[nr]^.Frecuencia);
    end;
```

Efecto 3 - Tone Portamento

En este efecto primero ha de determinar si la nota de destino es mayor o menor que la nota actual. En función de ello se ha de incrementar o decrementar la nota actual. La velocidad de la modificación se indica en el operando. La frecuencia se ha de modificar en cada tick, y fijarse de nuevo en la GUS. La inicialización para ello se incluye en la instrucción *Case* mediante un procedimiento propio.

```
3 : begin; { Nota Portamento }
      EI_toneportamento(nr);
    end;
{
 ---------------------------------------------
procedure EI_toneportamento(nr : byte);
{
 init para el procedimiento Vibrato (fuera del Effect-handling)
}
begin;
  { establecer factor de incremento }
  if Canales[nr]^.Operando <> 0 then
  begin;
    if Canales[nr]^.Start_Nota > Canales[nr]^.Ziel_Nota then
    begin;
      Canales[nr]^.slidespeed := -(Canales[nr]^.Operando);
    end else begin;
      Canales[nr]^.slidespeed := (Canales[nr]^.Operando);
    end;
  end;
  if Canales[nr]^.Start_Nota < 1 then
    Canales[nr]^.Start_Nota := 1;
  Canales[nr]^.Frecuencia :=
```

```
    longint(Voice_Base[14] div Canales[nr]^.Start_Nota);
  u_VoiceFreq(nr,Canales[nr]^.Frecuencia);
  oldv[nr] := Canales[nr]^.Start_Nota;
end;
 ---------------------------------------------
}
```

Efecto 4 - Vibrato

En el Vibrato ha de determinar primero la velocidad del Vibrato y su profundidad de los nibbles x e y del operando. Ahora se llama el procedimiento *Vibrato*. En él se incrementa un puntero en la tabla de Vibrato. Cuando sobrepasa *Ran* (>*64*), se decrementa en *64* colocándose así de nuevo en el rango adecuado. Con ayuda de este puntero se obtiene un valor de la tabla de Vibrato, que se ha de multiplicar con la profundidad de Vibrato y dividir entre *256*. En este valor se incrementa la frecuencia del tono. Este procedimiento se ha de llamar para cada tick y fijando la frecuencia de nuevo cada vez.

```
    4 : begin;  { Vibrato  *new* }
          Canales[nr]^.vibx := Canales[nr]^.Operando shr 4;
          Canales[nr]^.viby := Canales[nr]^.Operando and $0f;
          Efecto_vibrato(nr);
        end;
{
 ---------------------------------------------
procedure Efecto_vibrato(nr : byte);
{
 procedimiento Vibrato (fuera del Effect-handling)
}
var vibswap : integer;
begin;
  inc(Canales[nr]^.vibpos,Canales[nr]^.vibx);
  if Canales[nr]^.vibpos > 64 then
    dec(Canales[nr]^.vibpos,64);
  vibswap :=
    (VibratoTable[Canales[nr]^.vibpos] * Canales[nr]^.viby) div 256;
  inc(Canales[nr]^.Start_Nota,vibswap);
  if Canales[nr]^.Start_Nota < 1 then
    Canales[nr]^.Start_Nota := 1;
  Canales[nr]^.Frecuencia :=
    longint(Voice_Base[14] div Canales[nr]^.Start_Nota);
  u_VoiceFreq(nr,Canales[nr]^.Frecuencia);
end;
 ---------------------------------------------
}
```

Efecto 5 - Note & Volume sliding

Este efecto se puede partir en dos segmentos: por una parte el *Volume Sliding*, por otra el *Portamento*. Primero ha de comprobar si el operando es mayor de *0fh*. En este caso, existe un *Sliding down* y se ha de calcular la velocidad según *speed := Operando AND $0f*. De lo contrario nos encontramos con un *Sliding up*, y ha de desplazar (shift) el operando en cuatro posiciones para obtener la velocidad. Después de haber averiguado la velocidad, modifique primero el volumen de la voz. Tenga en cuenta que los valores no pueden abandonar los rangos permitidos. Lo mismo se aplica para la frecuencia. Deberá vigilar que la frecuencia no se aumente al infinito o quede menor que *1*. Ahora ya puede fijar la frecuencia y el volumen en la Gravis Ultrasound.

```
5 : begin; {NOTE SLIDE + VOLUME SLIDE:   *new* }
      { init }
      if Canales[nr]^.Operando <= $0f then
      begin;
        Canales[nr]^.vslide := -(Canales[nr]^.Operando AND $0f);
        Canales[nr]^.slidespeed:= -(Canales[nr]^.Operando AND $0f);
      end else begin;
        Canales[nr]^.vslide := (Canales[nr]^.Operando shr 4);
        Canales[nr]^.slidespeed := (Canales[nr]^.Operando shr 4);
      end;
      { volume slide }
      inc(Canales[nr]^.volume,Canales[nr]^.vslide);
      if Canales[nr]^.volume < 0 then Canales[nr]^.volume := 0;
      if Canales[nr]^.volume > 63 then Canales[nr]^.volume := 63;
      u_VoiceVolume(Nr,Canales[nr]^.volume);
      { Note slide }
      inc(Canales[nr]^.Start_Nota,Canales[nr]^.slidespeed);
      if Canales[nr]^.Start_Nota < 1 then
        Canales[nr]^.Start_Nota := 1;
      Canales[nr]^.Frecuencia :=
        longint(Voice_Base[14] div Canales[nr]^.Start_Nota);
      u_VoiceFreq(nr,Canales[nr]^.Frecuencia);
    end;
```

Efecto 6 - Vibrato & Volume sliding

También este efecto se puede dividir en dos subefectos. Para el *Volume sliding* se aplica lo mismo que detallamos bajo *Efecto 5* y para el *Vibrato* puede emplear las mismas rutinas que para el *Efecto Vibrato 4*.

```
6 : begin;  { Vibrato & Volume slide  *new* }
      { init }
      Canales[nr]^.vibx := Canales[nr]^.Operando shr 4;
      Canales[nr]^.viby := Canales[nr]^.Operando and $0f;
      if Canales[nr]^.Operando <= $0f then
      begin;
```

```
      Canales[nr]^.vslide := -(Canales[nr]^.Operando AND $0f);
    end else begin;
      Canales[nr]^.vslide := (Canales[nr]^.Operando shr 4);
    end;
    { volume slide }
    inc(Canales[nr]^.volume,Canales[nr]^.vslide);
    if Canales[nr]^.volume < 0 then Canales[nr]^.volume := 0;
    if Canales[nr]^.volume > 63 then Canales[nr]^.volume := 63;
    u_VoiceVolume(Nr,Canales[nr]^.volume);
    { vibrato }
    Efecto_vibrato(nr);
  end;
```

Efecto 7 - Tremolo

El efecto de *Tremolo* es idéntico en su funcionamiento que el *Vibrato*. Sólo que aquí se manipula el volumen y no la frecuencia.

```
7 : begin; { tremolo  *new* }
      Canales[nr]^.vibx := Canales[nr]^.Operando shr 4;
      Canales[nr]^.viby := Canales[nr]^.Operando and $0f;
      inc(Canales[nr]^.vibpos,Canales[nr]^.vibx);
      if Canales[nr]^.vibpos > 64 then
      dec(Canales[nr]^.vibpos);
      vibswap :=
     (VibratoTable[Canales[nr]^.vibpos]*Canales[nr]^.viby)div 256;
      inc(Canales[nr]^.Volume,vibswap);
      if Canales[nr]^.Volume < 0 then Canales[nr]^.Volume := 0;
      if Canales[nr]^.Volume > 63 then Canales[nr]^.Volume := 63;
      u_VoiceVolume(nr,Canales[nr]^.volume);
  end;
```

Efecto 8 - No empleado

Este efecto no se emplea. Sin embargo lo puede emplear muy bien para obtener efectos propios o sincronizaciones con los gráficos.

```
8 : begin; { not used !!! }
    {
      oficialmente no se emplea, por ello es muy adecuado para
      sincronizar algunos eventos en una demo...
    }
  end;
```

Efecto 9 - Sample-Offset

En el *Sample-Offset* se entiende el operando como offset en el sample. El byte designa el High-Byte del offset. Así que para obtener el verdadero, se ha de multipli-

car con *256*. Sume el offset *Inicial* a la posición del sample en la RAM de la Gravis Ultrasound y fije el canal en esta posición.

```
9 : begin; { Sampel - Offset *new* }
     swaplong := longint((Canales[nr]^.Operando+1)) * 256;
     Canales[nr]^.Mempos := Canales[nr]^.Mempos+swaplong;
    u_Voicedata(Canales[nr]^.Mempos,Canales[nr]^.Inicio_Bucle,
        Canales[nr]^.Fin,nr);
      U_StartVoice(nr,Play_Voice+Bit8+Canales[nr]^.Looping+Unidirect);
    end;
```

Efecto 0ah - Volume sliding

Este es uno de los efectos de más fácil realización. Si el operando tiene un valor menor de *16*, en cada tick ha de restar los 4 bits inferiores del operando del volumen actual. De lo contrario ha de desplazar el operando cuatro posiciones a la derecha y sumarlo en cada tick. Controle que el volumen se mantenga en el rango permitido.

```
$a : begin;  { Volume sliding  *new* }
      if Canales[nr]^.Operando <= $0f then
      begin;
        Canales[nr]^.vslide := -(Canales[nr]^.Operando AND $0f);
      end else begin;
        Canales[nr]^.vslide := (Canales[nr]^.Operando shr 4);
      end;
      inc(Canales[nr]^.volume,Canales[nr]^.vslide);
      if Canales[nr]^.volume < 0 then Canales[nr]^.volume := 0;
      if Canales[nr]^.volume > 63 then Canales[nr]^.volume := 63;
      u_VoiceVolume(Nr,Canales[nr]^.volume);
    end;
```

Efecto 0bh - Position Jump

Aquí el operando indica el número de la posición en el arreglo, a la que se ha de saltar. Simplemente fije la línea actual al final del patrón. Con ello obtiene que en la siguiente pasada se incremente automáticamente un patrón.

```
$b : begin; { Position Jump *ok* }
      runinf.Linea := 64;
      runinf.Pattnr := Canales[nr]^.Operando;
    end;
```

Efecto 0ch - Set Note Volume

Este efecto fija el volumen de una voz. Es suficiente con asignarle el valor del operando al volumen de la voz. Vigile que el volumen se mantenga en los rangos permitidos.

```
$c : begin; { Set Note Volume *ok* }
      if Canales[nr]^.Operando>63 then Canales[nr]^.Operando := 63;
      if Canales[nr]^.Operando < 1 then
      begin
        Canales[nr]^.volume := 0;
        u_VoiceVolume(nr,0);
        U_StartVoice(nr,Stop_Voice);
        stop_Thevoice[nr] := true;
      end else begin
        Canales[nr]^.volume := Canales[nr]^.Operando;
        u_VoiceVolume(Nr,Canales[nr]^.volume);
        Runinf.Volumes[nr] := 63;
      end;
    end;
```

Efecto 0dh - Pattern Break

Con este efecto puede saltar al siguiente patrón del MOD. Si quiere programar este efecto con toda corrección, ha de emplear una variable booleana, que siempre ha de estar en *TRUE* cuando se haya de saltar un patrón. Entonces puede emplear el valor indicado en el operando para la línea. Sin embargo también es suficiente si simplemente coloca la línea al final del patrón, de modo que se salte automáticamente al siguiente patrón.

```
$d : begin; { Patterm Break *ok* }
      runinf.Linea := 64;
    end;
```

Efecto 0eh - Comando de efecto ampliado

El efecto *0eh* define varios subefectos. El número del subefecto se obtiene de los cuatro bits superiores del operando. A nosotros nos interesarán los siguientes subefectos:

Subefecto 1 - Fine Sliding Up

Este efecto trabaja igual que el efecto de *Portamento up*. Sin embargo, aquí no se realiza ninguna actualización con cada tick, sino que la frecuencia sólo se cambia una vez durante la inicialización.

```
$e : begin;  { Comando de efecto ampliado }
       case (Canales[nr]^.Operando shr 4) of
         1 : begin; { Fine slide up }
```

```
                dec(Canales[nr]^.Start_Nota,Canales[nr]^.Operando and
$0f);
                if Canales[nr]^.Start_Nota < 1 then Canales[nr]
^.Start_Nota := 1;
                Canales[nr]^.Frecuencia :=
                  longint(Voice_Base[14] div Canales[nr]^.Start_Nota);
                u_VoiceFreq(nr,Canales[nr]^.Frecuencia);
              end;
```

Subefecto 2 - Fine Sliding Down

Se aplica lo mismo que en el subefecto *1*. La correspondencia está en *Portamento down*.

```
          2 : begin; { Fine slide down }
                inc(Canales[nr]^.Start_Nota,Canales[nr]^.Operando and
$0f);
                if Canales[nr]^.Start_Nota < 1 then Canales[nr]
^.Start_Nota := 1;
                Canales[nr]^.Frecuencia :=
                  longint(Voice_Base[14] div Canales[nr]^.Start_Nota);
                u_VoiceFreq(nr,Canales[nr]^.Frecuencia);
              end;
```

Subefecto 9 - Retriggering Note

En este efecto se vuelve a activar de nuevo la nota después de los ticks especificados en el operando. Esto quiere decir que el sample simplemente comienza de nuevo.

```
          9 : begin; { Retriggering !!! *new* }
                Canales[nr]^.Retrig_count :=
                  Canales[nr]^.Operando and $0f;
              end;
```

Subefecto 0ah - Fine Volume Slide Up

El volumen del canal se incrementa en el valor especificado en los cuatro bits inferiores. No se realiza ninguna actualización del volumen en los ticks.

```
          $a : begin; { fine volume slide up }
               Canales[nr]^.vslide := (Canales[nr]^.Operando AND $0f);
               inc(Canales[nr]^.volume,Canales[nr]^.vslide);
               if Canales[nr]^.volume<0 then Canales[nr]^.volume := 0;
               if Canales[nr]^.volume>63 then Canales[nr]^.volume:=63;
               u_VoiceVolume(Nr,Canales[nr]^.volume);
             end;
```

Subefecto 0bh - Fine Volume Slide Down

El volumen del canal se decrementa en el valor especificado en los cuatro bits inferiores. No se realiza ninguna actualización del volumen en los ticks.

```
$b : begin; { fine volume slide down }
      Canales[nr]^.vslide:= (Canales[nr]^.Operando AND $0f);
      dec(Canales[nr]^.volume,Canales[nr]^.vslide);
      if Canales[nr]^.volume<0 then Canales[nr]^.volume:= 0;
      if Canales[nr]^.volume>63 then Canales[nr]^.volume:=63;
      u_VoiceVolume(Nr,Canales[nr]^.volume);
     end;
```

Subefecto 0ch - Cut Voice

Con este efecto detiene la reproducción de un canal. En nuestro player simplemente hemos de colocar en *TRUE* la variable *Stop_TheVoice*.

```
$c : begin;  { Cut Voice *ok* }
      stop_Thevoice[nr] := true;
     end;
  end;
end;
```

Efecto 0fh - Set Speed

Con este efecto se ajusta la velocidad en el MOD. Si el operando tiene un valor menos de *16*, el dato se refiere a la *MODSpeed*, sino a los *BPM*. Si se trata de la *Speed*, se le asigna a la variable *ticklimit* el valor del operando. De lo contrario se fija el nuevo valor para los BPM mediante el procedimiento *nuevo_interrupt_speed*.

```
$f : begin; { Set Speed  *ok* }
      if Canales[nr]^.Operando <= $f then begin;
        ticklimit := Canales[nr]^.Operando;
        runinf.speed := ticklimit;
      end else begin;
        runinf.bpm := Canales[nr]^.Operando;
        Nueva_Vel_Interrup(Canales[nr]^.Operando);
      end;
     end;
  end;
end;
```

Los efectos naturalmente no sólo se han de inicializar, sino también actualizar en cada tick. Los algoritmos necesitados para ello son iguales en principio que los de la inicialización, sólo que habitualmente no es necesaria la inicialización de las variables. Aquí tiene el procedimiento responsable en un breve resumen:

```
procedure tick_effects;
var li : integer;
```

```
    vibswap : integer;
begin;
  for li := 1 to MOD_Voces do begin;
    if runinf.volumes[li] > 0 then
      dec(runinf.volumes[li]);
    case Canales[li]^.Efecto of { procesar efectos }
      0 : begin;
            inc(Canales[li]^.Appegpos);
            case (Canales[li]^.Appegpos MOD 3) of
              0 : begin; {ap = 3 !}
                    Canales[li]^.Start_Nota :=
                    Canales[li]^.Nota + Canales[li]^.Efectoy;
                  end;
              1 : begin; {ap = 1 !}
                    Canales[li]^.Start_Nota :=
                    Canales[li]^.Nota;
                  end;
              2 : begin; {ap = 2 !}
                    Canales[li]^.Start_Nota :=
                    Canales[li]^.Nota + Canales[li]^.Efectox;
                  end;
            end;
          end;
      1 : begin;
          {!"! new }
            Canales[li]^.Operando := Canales[li]^.Operando and $0F;
          {!"! new end }
            dec(Canales[li]^.Start_Nota,Canales[li]^.Operando);
            if Canales[li]^.Start_Nota < 1 then
              Canales[li]^.Start_Nota := 1;
            Canales[li]^.Frecuencia :=
              longint(Voice_Base[14] div Canales[li]^.Start_Nota);
            u_VoiceFreq(li,Canales[li]^.Frecuencia);
          end;
      2 : begin;
          {!"! new }
            Canales[li]^.Operando := Canales[li]^.Operando and $0F;
          {!"! new end }
            inc(Canales[li]^.Start_Nota,Canales[li]^.Operando);
            if Canales[li]^.Start_Nota < 1 then
              Canales[li]^.Start_Nota := 1;
            Canales[li]^.Frecuencia :=
              longint(Voice_Base[14] div Canales[li]^.Start_Nota);
            u_VoiceFreq(li,Canales[li]^.Frecuencia);
          end;
      3 : begin; { Notae Portamento }
            E_toneportamento(li);
{
 ---------------------------------------------------
procedure E_toneportamento(nr : byte);
```

```
{
 procedimiento NotaPortamento (fuera del Effect-handling)
}
begin;
  if Canales[nr]^.slidespeed < 0 then
  begin
    inc(Canales[nr]^.Start_Nota,Canales[nr]^.slidespeed);
    if Canales[nr]^.Start_Nota < Canales[nr]^.Ziel_Nota then
      Canales[nr]^.Start_Nota := Canales[nr]^.Ziel_Nota;
  end else begin
    inc(Canales[nr]^.Start_Nota,Canales[nr]^.slidespeed);
    if Canales[nr]^.Start_Nota > Canales[nr]^.Ziel_Nota then
      Canales[nr]^.Start_Nota := Canales[nr]^.Ziel_Nota;
  end;

  if Canales[nr]^.Start_Nota < 1 then
    Canales[nr]^.Start_Nota := 1;
  Canales[nr]^.Frecuencia :=
    longint(Voice_Base[14] div Canales[nr]^.Start_Nota);
  u_VoiceFreq(nr,Canales[nr]^.Frecuencia);
  oldv[nr] := Canales[nr]^.Start_Nota;
end;
 ---------------------------------------------------
}
          end;
      4 : begin;  { vibrato  *new* }
            Efecto_vibrato(li);
          end;
      5 : begin;
            { volume slide }
            inc(Canales[li]^.volume,Canales[li]^.vslide);
            if Canales[li]^.volume < 0 then Canales[li]^.volume := 0;
            if Canales[li]^.volume > 63 then Canales[li]^.volume:= 63;
            u_VoiceVolume(li,Canales[li]^.volume);
            { Note slide }
            inc(Canales[li]^.Start_Nota,Canales[li]^.slidespeed);
             if Canales[li]^.Start_Nota < 1 then
               Canales[li]^.Start_Nota := 1;
            Canales[li]^.Frecuencia :=
              longint(Voice_Base[14] div Canales[li]^.Start_Nota);
            u_VoiceFreq(li,Canales[li]^.Frecuencia);
          end;
      6 : begin;
            { volume slide }
            inc(Canales[li]^.volume,Canales[li]^.vslide);
            if Canales[li]^.volume < 0 then Canales[li]^.volume := 0;
            if Canales[li]^.volume > 63 then Canales[li]^.volume:= 63;
            u_VoiceVolume(li,Canales[li]^.volume);
            { vibrato }
            inc(Canales[li]^.vibpos,Canales[li]^.vibx);
            if Canales[li]^.vibpos > 64 then
```

```
            dec(Canales[li]^.vibpos);
          vibswap :=
          (VibratoTable[Canales[li]^.vibpos]*Canales[li]^.viby) div
256;
          inc(Canales[li]^.Start_Nota,vibswap);
           if Canales[li]^.Start_Nota < 1 then
             Canales[li]^.Start_Nota := 1;
          Canales[li]^.Frecuencia :=
            longint(Voice_Base[14] div Canales[li]^.Start_Nota);
          u_VoiceFreq(li,Canales[li]^.Frecuencia);
        end;
    7 : begin; { tremolo  *new* }
          inc(Canales[li]^.vibpos,Canales[li]^.vibx);
          if Canales[li]^.vibpos > 64 then
            dec(Canales[li]^.vibpos);
          vibswap :=
          (VibratoTable[Canales[li]^.vibpos]*Canales[li]^.viby) div
256;
          inc(Canales[li]^.Volume,vibswap);
          if Canales[li]^.Volume < 0 then Canales[li]^.Volume := 0;
          if Canales[li]^.Volume> 63 then Canales[li]^.volume:= 63;
          u_VoiceVolume(li,Canales[li]^.volume);
        end;
    8 : begin; { not used !!! }
        end;
    $a : begin; { Volume sliding   **new* }
          inc(Canales[li]^.volume,Canales[li]^.vslide);
          if Canales[li]^.volume < 0 then Canales[li]^.volume := 0;
          if Canales[li]^.volume > 63 then Canales[li]^.volume:= 63;
          u_VoiceVolume(li,Canales[li]^.volume);
        end;
    $e : begin;  { comando de efectos ampliado }
         case (Canales[li]^.Operando shr 4) of
            9: begin; { Retriggering !!! }
                 if Canales[li]^.Operando and $0f <> 0 then begin;
                   dec(Canales[li]^.Retrig_count);
                   if Canales[li]^.Retrig_count = 0 then begin;
                     Canales[li]^.Retrig_count :=
                          Canales[li]^.Operando and $0f;
                 u_Voicedata(Canales[li]^.Mempos,Canales[li]^.Inicio_Bucle,
                          Canales[li]^.Fin,li);
                U_StartVoice(li,Play_Voice+Bit8+Canales[li]^.Looping+Unidirect);
                   end;
                 end;
               end;
            end;
        end;
      end;
    end;
  end;
```

Estos han sido los procedimientos principales de la Unit *GUS_MOD*. Ahora sabe cómo puede cargar y reproducir un MOD. Sólo le resta por aprender cómo se puede volver eliminar el MOD de la memoria. Para ello puede hacer servir el procedimiento *_gus_termina_mod*. Este llama primeramente el procedimiento *restaurar_timer*, que ya conoce. A continuación se libera la memoria ocupada mediante el procedimiento *dispose_mod*. Esta rutina detiene todos los canales de la Gravis Ultrasound. Después libera en un bucle toda la memoria ocupada por los patrones, las informaciones de canales y de instrumentos.

```
procedure dispose_mod;
{
 Elimina un MOD cargado de la memoria. Los samplings de la GUS NO se
borran
}
begin;
  for i := 0 to 31 do begin;
    U_StartVoice(i,Stop_Voice);
  end;
  for i := 1 to Vh.Num_Patts do begin;
    dos_freemem(Pattern[i]);
  end;
  for i := 0 to 15 do begin;
    dispose(Canales[i]);
  end;
  for i := 0 to 31 do begin;
    dispose(Instrumentos[i]);
  end;
end;

procedure _gus_terminamod;
{
 termina la reproducción de un MOD
}
begin;
  Reponer_IntTimer;
  dispose_mod;
end;
```

Esto es todo lo que ha de saber para programar un MOD-Player con la GUS. Les deseamos mucha diversión durante sus ensayos y modificaciones. ¿Qué tal una pista de efectos de sonido, o un MOD-Player en modo gráfico?

El Freak-GUS-Player TCP de «The CoExistence»

De momento nos conformaremos con un GUS-Player en modo texto. Para poder colocar más informaciones en la pantalla, emplearemos el modo de 50 líneas.

El modo de funcionamiento es bastante sencillo. Primero se inicializa la Gravis Ultrasound mediante *_gus_init_env*. Si no se encontró ninguna GUS, o si apareció un error durante la inicialización, se muestra un texto de ayuda. Este texto se creó con *The Draw* y está guardado en el formato Pascal-Object. Este formato es una simple imagen de la memoria de pantalla. Puede linkar el archivo objeto correspondiente en su programa de Pascal y direccionar la imagen con el procedimiento indicado. Si ha guardado por ejemplo el texto a mostrar bajo el nombre de procedimiento *helptxt*, puede mostrar la pantalla ANSI con

```
move (@helptxt^, ptr($B800,0)^, 80 * 50 * 2)
```

Si la inicialización tuvo éxito, se reserva la memoria para la estructura de manejo de archivos. Esta es parte de la Unit *fselect*, que puede encontrar en el CD adjunto. Ahora se comprueba si se especificó un nombre de archivo en la línea de comandos. Esto lo realiza la función *check_commandline*. Devuelve el valor *TRUE* si se especificó un nombre de archivo. De lo contrario podrá elegir uno o varios archivos mediante la Unit *fselect*.

Ahora se realizan algunos preparativos para la Unit MOD mediante *_gus_inicializar*. A continuación se muestra el *ANSI* para la pantalla principal. Después se llama el procedimiento *write_phunliners*. Este procedimiento lee tres líneas del archivo de texto PHUN.TXT y las representa en la zona inferior de la pantalla. Las entradas del archivo PHUN.TXT es de *BBS-On-liner* y libremente editable.

Después de la visualización de los Phunliners se carga el archivo MOD mediante *_gus_modload*. Esta función ya la conoce extensamente. Después de haber cargado el MOD, se representan en pantalla el nombre de archivo del MOD, así como todas las informaciones importantes (inombres de instrumento!). Una vez finalizadas las operaciones gráficas, se inicia la reproducción del MOD mediante *_gus_iniciarmod*.

Ahora se llama *User*. Este procedimiento actualiza regularmente la pantalla (Volume Bars) y se encarga simultáneamente de las entradas del usuario. Cuando se abandona, se elimina el MOD actual de la memoria. Según si el usuario eligió el archivo al principio o desde el menú de selección de archivos, volverá a este menú o bien se termina el player. El coding en sí del player es trivial, fundamentalmente se apoya en la Unit *GUS_MOD*. Aquí tiene su código:

```
{$A+,B-,D+,E+,F+,G+,I+,L+,N-,O-,P-,Q-,R-,S+,T-,V+,X+,Y+}
{$M 16384,0,250000}

program gusdemo;

uses crt,dos,design,fselect, gus_mod;

type
  Pphun = ^TPhun;
  TPhun = array[0..799] of string[80];

const Ruta_MOD = 'd:\mods';

const Terminar_Programa : boolean = false;

var phun : Pphun;
    phuncount : integer;
    modify_voice : integer;
    i : integer;
    Los_Archivos : Pfileselect_struct;
    curr_modnr : integer;

{
  incluir las pantallas ANSI
}

{$L tcpans}
procedure tcpans; external;
{$L we_are}
procedure we_are; external;
{$L buy_it}
procedure buy_it; external;
{$L call}
procedure call; external;
{$L helptxt}
procedure helptxt; external;


function Archivo_existe(dname : string) : boolean;
{
 comprueba si el archivo pasado existe
}
var dumf : file;
begin;
  {$I-}
  assign(dumf,dname);
  reset(dumf,1);
  {$I+}
  if IOResult <> 0 then
```

```
    Archivo_existe := false
  else begin;
    Archivo_existe := true;
    close(dumf);
  end;
end;

procedure color_writeln(s : string);
{
 para visualizar una cadena en la combinación de colores TC
}
var colpos,li : integer;
begin;
  colpos := 1;
  for li := 1 to length(s) do begin;
    if s[li] = ' ' then colpos := 0;
    inc(colpos);
    case colpos of
        1..2 : begin;
                 textcolor(8);
               end;
        3..4 : begin;
                 textcolor(2);
               end;
      5..$ff : begin;
                 textcolor(10);
               end;
    end;
    write(s[li]);
  end;
end;

procedure Escribe_Nombre_Archivo(s : string);
{
 visualiza centrado el nombre de archivo del tema
}
var li,slen : integer;
begin;
  gotoxy(33,13);
  while pos('\',s) <> 0 do begin;
    delete(s,1,pos('\',s));
  end;
  slen := length(s);
  slen := (15 - slen) div 2;
  for li := 1 to slen do s := ' '+s;
  write(s);
end;

procedure Escribe_phunliners;
```

```
{
 Lee tres líneas del archivo «PHUN.TXT», y las visualiza en pantalla.
 El archivo «PHUN.TXT» es un archivo de texto libremente editable,
 que puede modificar a su antojo!
}
var tf : text;
begin;
  randomize;
  if not Archivo_existe('phun.txt') then exit;
  assign(tf,'phun.txt');
  reset(tf);
  phuncount := 0;
  {$I+}
  if ioresult = 0 then begin;
    while not eof(tf) do begin;
      readln(tf,phun^[phuncount]);
      inc(phuncount);
    end;
    close(tf);
    gotoxy(3,43);
    color_writeln(phun^[random(phuncount)]);
    gotoxy(3,44);
    color_writeln(phun^[random(phuncount)]);
    gotoxy(3,45);
    color_writeln(phun^[random(phuncount)]);
  end;
  {$I-}
end;

procedure display_modinfos;
{
 visualiza los nombre de instrumento del módulo actual
}
var li : integer;
begin;
  textcolor(14);
  textbackground(black);
  for li := 1 to 16 do begin;
    gotoxy(6,17+li);
    color_writeln(Instrumentos[li]^.name);
    gotoxy(50,17+li);
    color_writeln(Instrumentos[li+16]^.name);
  end;
end;

procedure exit_program;
{
 antes de terminar del programa un breve aviso al TC WHQ,
 la Farpoint Station (04202 76145)
```

```
}
begin;
  display_ansi(@call,co80+font8x8);
  cursor_off;
  repeat until keypressed;
  while keypressed do readkey;
  cursor_on;
  asm mov ax,03; int 10h; end;
  halt;
end;

procedure siguiente_MOD;
{
 comienza la reproducción del siguiente MOD seleccionado
}
begin;
  _gus_terminamod;
  inc(curr_modnr);
  if curr_modnr > Los_Archivos^.nofiles then
    curr_modnr := 1;
  if not _gus_cargamod(Los_Archivos^.fn[curr_modnr]) then begin;
    clrscr;
    gotoxy(10,10);
    write('Sorry dude, Cant''t handle this MOD-File');
    delay(1200);
    exit_program;
  end;
  display_ansi(@tcpans,co80+font8x8);
  cursor_off;
  Escribe_phunliners;
  Escribe_Nombre_Archivo(Los_Archivos^.fn[curr_modnr]);
  display_modinfos;
  fillchar(Play_Chanel,14,1);
  _gus_iniciamod;
end;

procedure display_we_are;
{
 visualiza un ANSI con infos sobre el grupo THE COEXISTENCE
}
begin;
  display_ansi(@we_are,co80+font8x8);
  cursor_off;
  repeat until keypressed;
  while keypressed do readkey;
  display_ansi(@tcpans,co80+font8x8);
  cursor_off;
  Escribe_phunliners;
  Escribe_Nombre_Archivo(Los_Archivos^.fn[curr_modnr]);
```

```
  display_modinfos;
end;

procedure display_buy_it;
{
 visualiza publicidad para el libro PC al límite
}
begin;
  display_ansi(@buy_it,co80+font8x8);
  cursor_off;
  repeat until keypressed;
  while keypressed do readkey;
  display_ansi(@tcpans,co80+font8x8);
  cursor_off;
  Escribe_phunliners;
  Escribe_Nombre_Archivo(Los_Archivos^.fn[curr_modnr]);
  display_modinfos;
end;

procedure handle_keys(key1,key2 : char);
{
 reacciona a las entradas de teclado del usuario
}
var pchan : byte;
begin;
  case key1 of
    #00 : begin;
            case key2 of
              #45 : begin;
                      Terminar_Programa := true;
                    end;
              #72 : begin;
                      if modify_voice > 1 then
                        dec(modify_voice);
                    end;
              #80 : begin;
                      if modify_voice < Modinf.Voces then
                        inc(modify_voice);
                    end;
              #75 : begin; { cursor left }
                      runinf.Linea := 64;
                      dec(runinf.Pattnr,2);
                      if runinf.Pattnr < -1 then runinf.Pattnr := -1;
                    end;
              #77 : begin; { cursor right }
                      runinf.Linea := 64;
                      inc(runinf.Pattnr);
                    end;
            end;
```

```
      end;
#27 : begin;
        Terminar_Programa := true;
      end;
#32,
'W',
'w',
'I',
'i' : begin;
        display_we_are;
      end;
'D',
'd',
'b',
'B' : begin;
        display_buy_it;
      end;
'L',
'l' : begin;
        chpos[modify_voice] := 1;
        _gus_set_chanelpos;
      end;
'R',
'r' : begin;
        chpos[modify_voice] := 15;
        _gus_set_chanelpos;
      end;
'M',
'm' : begin;
        chpos[modify_voice] := 7;
        _gus_set_chanelpos;
      end;
'U',
'u' : begin;
        if Modinf.Voces = 4 then
        begin
          chpos[1] := 2;
          chpos[2] := 5;
          chpos[3] := 9;
          chpos[4] := 12;
        end;
        if Modinf.Voces = 8 then
        begin
          chpos[1] := 1;
          chpos[2] := 3;
          chpos[3] := 5;
          chpos[4] := 7;
          chpos[5] := 7;
          chpos[6] := 9;
```

```
            chpos[7] := 11;
            chpos[8] := 13;
          end;
          _gus_set_chanelpos;
        end;
    ',' : begin; { a la izquierda }
          if chpos[modify_voice] > 1 then
            dec(chpos[modify_voice]);
          _gus_set_chanelpos;
        end;
    '.' : begin; { a la derecha }
          if chpos[modify_voice] < 15 then
            inc(chpos[modify_voice]);
          _gus_set_chanelpos;
        end;
    '1'..
    '8' : begin;
          pchan := ord(key1)-48;
          if Play_Chanel[pchan] = 1 then begin;
            Play_Chanel[pchan] := 0;
            textcolor(10);   gotoxy(77,2+pchan);
            write('M');      textcolor(2);
            write('UTE');
          end else begin;
            Play_Chanel[pchan] := 1;
            textcolor(10);   gotoxy(77,2+pchan);
            write('C');      textcolor(2);
            write('H  ');
          end;
        end;
    'n',
    'N' : begin;
          siguiente_MOD;
        end;
  end;
end;

procedure screen_update;
const colvals : array[1..35] of byte =
   (08,08,08,08,08,02,02,02,02,10,10,10,10,10,10,10,10,
    10,10,10,10,10,10,10,10,10,10,10,10,10,05,05,05,05,05);
var volstr : string[66];
    li : integer;
    niv : integer;
begin;
  { actualizar barras de volumen}
  for li := 1 to Modinf.Voces do begin;
    for niv := 1 to round(Runinf.Volumes[li] / 1.78) do begin;
      screen[li+2,37+niv].a := colvals[niv];
```

```
    end;
    for niv := round(Runinf.Volumes[li] / 1.78)  to 36 do begin;
      screen[li+2,38+niv].a := 7;
    end;
  end;

  { colocar color de fondo correcto para la flecha }
  for li := 1 to 8 do begin;
    if li = modify_voice then begin;
      screen[2+li,34].a := 05;
      screen[2+li,35].a := 05;
      screen[2+li,36].a := 05;
      screen[2+li,37].a := 05;
    end else begin;
      screen[2+li,34].a := 07;
      screen[2+li,35].a := 07;
      screen[2+li,36].a := 07;
      screen[2+li,37].a := 07;
    end;
  end;

  { visualizar informaciones de reproducción de los MOD}
  gotoxy(18,14);
  color_writeln(Modinf.Titulo);

  textcolor(7);
  gotoxy(18,16);
  write(runinf.pattnr:3);

  gotoxy(64,16);
  write(runinf.Linea:3);

  gotoxy(64,15);
  write(64:3);

  gotoxy(18,15);
  write(modinf.Patt_anz:3);

  gotoxy(60,14);
  write(runinf.speed,' / ',runinf.bpm);
end;

procedure user;
{
 comprueba las entradas de teclado, y actualiza la pantalla
}
var ch1,ch2 : char;
begin;
  repeat
```

```
    ch1 := #255;
    ch2 := #255;
    if keypressed then begin;
      ch1 := readkey;
      if keypressed then ch2 := readkey;
      handle_keys(ch1,ch2);
    end;
   screen_update;
  until Terminar_Programa;
end;


procedure display_help;
{
 Visualiza el ANSI de ayuda. También se muestra si no se encuentra
GUS
}
begin;
  display_ansi(@helptxt,co80+font8x8);
  cursor_off;
  repeat until keypressed;
  while keypressed do readkey;
  exit_program;
end;


function check_commandline : boolean;
{
 returns true, if a module-name was given
}
var pst : string;
    ist_mod : boolean;
    li : integer;
    retval : boolean;
begin;
  retval := false;
  for li := 1 to 9 do begin;
    pst := paramstr(li);
    ist_mod := true;

    if (pos('-h',pst) <> 0) or (pos('-H',pst) <> 0) or
       (pos('-?',pst) <> 0) then
    begin;
      ist_mod := false;
      display_help;
    end;

    if (pst <> '') and ist_mod then begin;
      if pos('.',pst) = 0 then pst := pst + '.mod';
      if Archivo_existe(pst) then  { gült. mod }
```

```
      begin
        inc(Los_Archivos^.nofiles);
        Los_Archivos^.fn[Los_Archivos^.nofiles] := pst;
        retval := true;
      end;
    end;
  end;
  check_commandline := retval;
end;

begin;
  cursor_off;
  clrscr;
  if not _gus_init_env then display_help;
  new(Los_Archivos);
  new(phun);
  Los_Archivos^.path := Ruta_MOD;
  Los_Archivos^.Mask := '*.mod';
  Los_Archivos^.sx   := 24;
  Los_Archivos^.sy   := 10;
  Los_Archivos^.nofiles := 0;

  Los_Archivos^.Titulo := 'Seleccionar archivo MOD!!!';
  modify_voice := 1;

  for i := 1 to 30 do
    Los_Archivos^.fn[i] := '---';
  save_screen;
  if not check_commandline then begin;
    Selecc_ArchComp(Los_Archivos);
    repeat
      restore_screen;

      if Los_Archivos^.fn[1] = '---' then exit_program;

        _gus_inicializar;

      display_ansi(@tcpans,co80+font8x8);
      cursor_off;
      Escribe_phunliners;

      curr_modnr := 1;
      if not _gus_cargamod(Los_Archivos^.fn[1]) then begin;
        clrscr;
        gotoxy(10,10);
        write('Sorry dude, Cant''t handle this MOD-File');
        delay(1200);
        exit_program;
```

```
      end;
      Escribe_Nombre_Archivo(Los_Archivos^.fn[1]);
      display_modinfos;
      fillchar(Play_Chanel,14,1);
      _gus_iniciamod;

      user;
      _gus_terminamod;
      dispose(Los_Archivos);
      new(Los_Archivos);
      Los_Archivos^.path := Mod_pfad;
      Los_Archivos^.Mask := '*.mod';
      Los_Archivos^.sx   := 24;
      Los_Archivos^.sy   := 10;
      Los_Archivos^.nofiles := 0;
      for i := 1 to 30 do
        Los_Archivos^.fn[i] := '---';
      Terminar_Programa := false;
      Selecc_ArchComp(Los_Archivos);
    until Los_Archivos^.fn[1] = '---';

      dispose(Los_Archivos);
    dispose(phun);
    exit_program;
  end else begin;
  restore_screen;

  if Los_Archivos^.fn[1] = '-' then exit_program;

  _gus_inicializar;

  display_ansi(@tcpans,co80+font8x8);
  cursor_off;
  Escribe_phunliners;

  curr_modnr := 1;
  _gus_cargamod(Los_Archivos^.fn[1]);
  Escribe_Nombre_Archivo(Los_Archivos^.fn[1]);
  display_modinfos;
  fillchar(Play_Chanel,14,1);
  _gus_iniciamod;

  user;

  _gus_terminamod;
  dispose(Los_Archivos);
  dispose(phun);
  exit_program;

  end;
end.
```

16. Útiles herramientas de Shareware

En el siguiente capítulo les presentamos algunas utilidades muy buenas, que como Freak le harán un buen servicio. Así, desde el ensamblador A86 hasta el programa de rendering Povray podrá encontrar una buena selección de estupendo Shareware.

Para aquellos entre ustedes a los que les guste emplear los programas de Windows, también se han incluido los visualizadores gráficos Graphics WorkShop y Paintshop Pro.

N. del T.: Ante la coincidencia en castellano de los términos ingleses *file* (archivo) y *archive* (archivo), en los siguientes apartados se empleará el término *bloque de archivos* para designar un conjunto de *archivos* empaquetados por un programa compresor.

16.1 Recupere espacio en el disco - ARJ 2.41

ARJ, al igual que PKZIP, es un programa de compresión de datos. Probablemente es el formato de compresión más popular después de PKZIP. Los datos comprimidos con él tienen la extensión ARJ, o si ha dividido los datos en varios disquetes, también A01, A02, A03, etc.

El empleo de ARJ

Si escribe ARJ en la línea de comandos del DOS, obtendrá la siguiente pantalla de ayuda:

```
ARJ 2.41a Copyright (c) 1990-93 Robert K Jung. Jul 10 1993
*** This SHAREWARE program is NOT REGISTERED for use in a business,
commercial,*** government, or institutional environment except for
evaluation purposes.

List of frequently used commands and switches. Type ARJ -? for more help

Usage:    ARJ <command> [-<sw> [-<sw>...]] <archive_name>
[<file_names>...]
Examples: ARJ a -e archive, ARJ e archive, ARJ l archive *.doc

<Commands>
  a: Add files to archive          m: Move files to archive
  d: Delete files from archive     t: Test integrity of archive
  e: Extract files from archive    u: Update files to archive
```

```
  f: Freshen files in archive          v: Verbosely list contents of archive
  l: List contents of archive          x: eXtract files with full pathname
<Switches>
  r: Recurse subdirectories            e: Exclude paths from names
  g: Garble with password              v: enable multiple Volumes
  y: assume Yes on all queries
```

Los diferentes parámetros tienen los siguientes significados:

a: Add files to archive

Con esta función puede crear un bloque de archivos, o añadir archivos a un bloque de archivos.

```
C:\> ARJ a TEST.ARJ C:\TEST\*.*
```

Con esto se añaden todos los archivos del directorio TEST al bloque de archivos TEST.ARJ. Si el bloque de archivos TEST.ARJ no existe, será creado automáticamente.

m: Move files to archive

Esta función permite acoger archivos al bloque de archivos, que serán borrados del disco después de añadirlos con éxito.

```
C:\> ARJ m TEST.ARJ C:\TEST\*.*
```

Todos los archivos se borrarán, una vez que se hayan añadido al bloque de archivos.

d: Delete files from archive

ARJ también permite el borrado de archivos de un bloque de archivos. Para ello emplee el parámetro *d*.

```
C:\> ARJ d TEST.ARJ TEST.TXT
```

Se borrará el archivo TEST.TXT del bloque de archivos TEST.ARJ. También puede emplear los llamados comodines para borrar varios archivos que presenten características similares:

```
C:\> ARJ d TEST.ARJ *.TXT
```

t: Test integrity of archive

Con el parámetro *t* puede comprobar la integridad (el estado) de un bloque de archivos ARJ. ARJ comprobará si el bloque de archivos está destruido o dañado, y avisará con mensajes de error correspondientes.

```
C:\> ARJ t TEST.ARJ
```

Si el bloque de archivos está dañado, ARJ muestra los errores, y podrá actuar en consecuencia para reparar el bloque de archivos ARJ.

e: Extract files from archive

Este comando desempaquetará uno o varios archivos al directorio actual (o al indicado). ARJ le preguntará, cuando sea necesario, si quiere sobrescribir posibles archivos existentes. Si contesta con *N* de *NO*, ARJ le preguntará un nuevo nombre para el archivo a desempaquetar:

```
C:\> ARJ e TEST.ARJ C:\TEST *.TXT
```

La línea de comandos provoca el desempaquetado de todos los archivos del bloque de archivos TEST.ARJ, que terminen en TXT, al directorio C:\TEST.

u: Update files to archive

Renueve los archivos antiguos con nuevos, y añada archivos nuevos con *u*.

```
C:\> ARJ u TEST.ARJ
```

f: Freshen files in archive

Mediante la función *f* renueva los archivos en un bloque de archivos de manera similar a la función *Update*. Si embargo, aquí sólo se renovarán los archivos que sean más viejos que los archivos seleccionados del soporte de datos.

```
C:\> ARJ f TEST.ARJ *.TXT *.DOC
```

Durante el refresco de archivos debería emplear las mismas funciones que utilizó para la creación del bloque de archivos:

```
C:\> ARJ a TEST.ARJ C:\TEST\ *.* -r
C:\> ARJ f TEST.ARJ C:\TEST\ *.* -r
```

v: Verbosely list contents of archive

Este comando muestra el contenido del bloque de archivos, con nombres de directorio completos, así como los comentarios del archivo.

```
C:\> ARJ v TEST.ARJ
```

l: List contents of archive

También puede listar el contenido del bloque de archivos con el comando *l*. Sin embargo, a diferencia de la función *v* (véase antes) los archivos se muestran en el orden en que se almacenaron.

```
C:\> ARJ l TEST.ARJ -jp
```

Si emplea este comando, así como el comando *v* con la opción *-jp*, después de cada página de pantalla se incluirá una pausa.

x: eXtract files with full pathname

Desempaqueta uno o varios archivos de un bloque de archivos en sus directorios, que anteriormente se guardaron en el bloque de archivos.

```
C:\> ARJ x TEST.ARJ
```

r: Recurse subdirectories

Con este parámetro puede incluir todos los subdirectorio en un bloque de archivos comprimido, o los archivos especificados con comodines que se encuentren en los subdirectorios.

e: Exclude paths from names

Esta opción permite el no guardar los nombres de directorio en el bloque de archivos. Así se impide que al desempaquetar se pueda ver en qué directorio se guardaron los archivos.

g: Garble with password

Asegure sus bloques de archivos con una clave mediante el parámetro *g*. Así puede echar un cerrojo a un acceso no autorizado.

```
C:\>ARJ a TEST.ARJ -gTEST
```

v: enable multiple Volumes

Esta opción es una de las más ventajosas del programa compresor ARJ. Con ella puede crear los llamados volúmenes múltiples. Es decir, puede crear un bloque de archivos dividido en varios bloques de archivos, para distribuirlo por ejemplo en sendos disquetes.

```
C:\>ARJ a -v720 TEST.000
```

Aquí se crearán bloques de archivos que no sean mayores de 720 KBytes.

y: assume Yes on all queries

ARJ pregunta, por ejemplo antes de sobrescribir un archivo, si desea sobrescribirlo o no. Si quiere obviar estas consultas, simplemente emplee la opción *y*.

```
C:\>ARJ e -y TEST.ARJ
```

ARJ - una alternativa a PKZIP?

Seguramente ARJ es uno de los mejores programa compresores, sin embargo presenta desventajas con respecto a PKZIP:

- ARJ no soporta memoria EMS/XMS, de modo que no puede trabajar con ARJ en caso de falta de memoria.
- ARJ no reconoce ni emplea DPMI (DOS Protected Mode Interface).

De modo que ARJ es uno de los programas compresores más destacados, pero lamentablemente sólo la segunda opción después de PKZIP. Sólo si ha de colocar un bloque de archivos en varios disquetes, podemos aconsejarle ARJ.

16.2 Ahorre espacio con PKLite

El compresor de programas PKLITE es una utilidad muy buena. Es Shareware, y se puede emplear para proyectos no comerciales. Comprime archivos EXE, pero dejándolos en forma ejecutable. Los programas se descomprimen en tiempo de ejecución.

PKLite, con respecto a su velocidad y el tamaño de los programas generados, es claramente el líder de mercado. La mayoría de grandes empresas de software emplean la versión comercial, para que sus programas no aparezcan tan gigantescos. PKLite se llama de la siguiente forma:

```
PKLITE [Opciones] archivo original [archivo destino].
```

El archivo original es el nombre del archivo EXE a comprimir. Si el archivo comprimido ha de llevar el mismo nombre, omita el último parámetro. De lo contrario puede especificar aquí el nombre del nuevo programa. Además dispone de las siguientes opciones:

-a Los archivos con Overlays internos se comprimen siempre. Además se realiza una optimización del realojamiento.

-b Al especificar esta opción, PKLite crea automáticamente un archivo de copia de seguridad, antes de sobrescribir un archivo. Si no está seguro de que el programa deseado se dejará comprimir sin problemas, debería emplear esta opción.

-e En la versión comercial puede especificar mediante esta opción si se debe emplear un método de compresión especial. Los archivos comprimidos de esta forma ya no se podrán descomprimir con PKLite.

-l Si le interesan las condiciones de licencia de PKLite, puede verlas mediante esta opción.

-n Al especificar este parámetro no se comprimirán los archivos de programa que contienen Overlay. El programa tampoco intenta optimizar el realojamiento.

-o Este comando es aproximadamente la inversión del comando *-b*. Si especifica este parámetro, se sobrescribirán todos los archivos existentes con el nombre correspondiente. PKLite ya no preguntará.

-r Esta opción se ha de disfrutar con cuidado. Corta todos los datos del final regular de un archivo EXE. Estos pueden ser datos de configuración, pero también datos de Overlay. Sólo debería emplear esta opción, si está realmente seguro de lo que hace.

-u De forma estándar, PKLite emplea la fecha y hora del archivo original. Si quiere que se empleen la fecha y hora actuales, debería emplear la opción *-u*.

-x Para descomprimir un archivo empaquetado, debería emplear esta opción. Esto puede llegar a ser útil, si busca informaciones determinadas, imágenes, etc., y el programa no funciona correctamente en su versión comprimida.

Si durante la compresión del archivo ha aparecido un error, PKLite genera uno de los siguientes mensajes y códigos de error:

Syntax Error

Este error aparece, cuando ha cometido un fallo en la línea de comandos. Probablemente ha introducido erróneamente el nombre del archivo, o ha omitido el guión antes de una opción. PKLite le volverá a mostrar la pantalla con todas las opciones.

File extension must be EXE or COM

PKLite sólo comprime archivos con las extensiones .EXE y .COM. Si su archivo tiene otra extensión, y sabe seguro que es un archivo ejecutable, puede renombrarlo como .EXE y comprimirlo.

Cannot open input file

Este mensaje de error puede tener varias causas. Bien su disco/disquete tiene un error, o está trabajando en red, o el archivo está siendo empleado por otra aplicación. En este caso, el archivo se bloquea.

Could not open output file

Para este mensaje también hay diferentes causas. Bien su disco duro está defectuoso, o está bloqueado por otra aplicación. Pero también puede ocurrir que haya especificado una vía de acceso errónea, y por ello PKLite no puede crear el archivo.

Write error

Este error aparece cuando el disco/disquete en el que se quiere escribir está defectuoso, o el archivo está bloqueado por otra aplicación.

Disk full error

Este mensaje de error aparece cuando no queda espacio suficiente en el disco. PKLite sólo puede borrar el archivo antiguo cuando ha terminado de escribir el nuevo. De modo que necesita espacio para ambos archivos en el disco.

Read error

Ha aparecido un error al leer el archivo antiguo.

Create error

PKLite no pudo crear el archivo especificado. La causa puede ser que haya especificado un directorio erróneo, o que el disco/disquete estén llenos.

Memory error

PKLite dispone de poca memoria principal para la ejecución. El programa necesita al menos 85 KBytes de memoria libre.

Cannot compress file into itself

Este mensaje aparece siempre que especifique como segundo parámetro el mismo nombre del archivo original. Si quiere emplear el mismo nombre, simplemente omita el segundo parámetro. De lo contrario hay que indicar un nombre distinto.

EXE header error

Este mensaje significa que la cabecera EXE contiene demasiadas informaciones como para ser comprimida. En la mayoría de los casos basta con proporcionarle más memoria libre al programa.

No extract code error

Este error aparece cuando intenta desempaquetar un archivo mediante la opción *-x*. Una razón para ello puede ser que el archivo no se haya comprimido con PKLite. También puede ser que se eliminó la identificación de descompresión.

Data error

Este error se provoca cuando PKLite detecta un error en el archivo comprimido. El archivo comprimido es corrupto.

Compressing many files into one file error

PKLite no puede comprimir varios archivos en uno solo. Por ejemplo ha intentado ejecutar *PKLite *.exe hallo.exe*.

Output wildcards error

Ha introducido un comodín para el archivo destino (* o ?). Esto no está permitido para el nombre del archivo de destino.

16.3 Rey de los marcadores con Game Wizard

Game Wizard es un programa TSR que le permite buscar determinados contenidos de la memoria, y modificarlos. De modo que es una especie de entrenador universal. La versión shareware ofrece algunas limitaciones con respecto de la

versión registrada. En esta puede guardar Memory-Tables completas, una vez que las haya encontrado. Con ello sólo es necesario buscar una sola vez las variables correspondientes de su juego preferido y cargarlas una y otra vez, cuando juegue con él. También puede intercambiar estas tablas con sus conocidos, y hacer así una rápida colección para todos los juegos corrientes.

La llamada de Game Wizard

Copie simplemente todos los archivos del directorio Game-Wizard a un subdirectorio de su disco duro. El programa se puede ejecutar mediante la introducción de *GWSHARE* en la línea de comandos. Si no le basta la breve explicación del libro, también puede ejecutar el estupendo programa tutorial de Game Wizard. Se llama *GWTUTOR* y se encuentra en el mismo directorio. Le llevará paso a paso a través del manejo de Game Wizard.

Una vez ejecutado Game Wizard, se instala en memoria de forma residente. Puede activarlo con la tecla [[ ], la tecla a la derecha de la [P].

El manejo de Game Wizard

A continuación le familiarizaremos con el manejo de Game Wizard. Le presentaremos paso a paso los distintos puntos de menú, que se encuentran dispuestos de forma lógica, según su empleo.

Memory Address Search

Esta opción le permite registrar la memoria empleada por su programa en busca de variables como dinero, vidas, municiones. Inicia la búsqueda introduciendo simplemente el valor a buscar en la ventana de entrada y pulsando [Return]. Además dispone de las siguientes teclas:

Teclas	Significado
[Ctrl]+[E]	Pulse esta combinación de teclas si quiere comenzar la búsqueda de una nueva entrada de memoria.
[Ctrl]+[P]	Mediante esta combinación de teclas puede ir un paso hacia atrás, si en la búsqueda anterior ha introducido un valor erróneo. Sín embargo, sólo puede retroceder un valor, no varios. ¿Y cómo se realiza en concreto la búsqueda de una variable?

Pongamos el caso de que al principio del juego posee 1000 monedas de oro. Ahora activa Game Wizard y selecciona el punto de menú *Memory Address Search*, introduce 1000, y pulsa [Return]. En el menú principal pulsa a continuación [ESC] para volver al juego. Ahora juegue hasta que haya gastado dinero. Si por ejemplo sólo

le quedan 765 piezas de oro, cambie de nuevo a Game Wizard, e introduzca el nuevo valor 765. Si Game Wizard ha encontrado la posición de memoria buscada, se le mostrará el resultado de la acción de búsqueda: la dirección en la que se encuentra la variable. De lo contrario habrá de continuar buscándola.

Table of Memory Locations

Después de haber encontrado la variable buscada, la puede "congelar", o modificar. Congelar significa que esta variable ya no se podrá modificar. En pantalla verá cuatro columnas por cada valor encontrado. En la primera columna puede elegir si la variable ha de ser congelada. En la segunda puede introducir una descripción para identificar la variable. La tercera columna contiene la posición de la variable en memoria y la cuarta contiene su valor. Para editar las entradas dispone de las siguientes teclas:

Tecla	Significado	
0	Con esto puede modificar el valor de la variable.	
C	Para borrar una entrada de la tabla pulse la tecla "C".	
E	Mediante "E" pasa al modo de edición. En él dispone de las siguientes teclas:	
	Tab	Salta a la siguiente columna.
	Ctrl + S	Para introducir la dirección deseada.
	Return	Guarda las modificaciones realizadas.
	ESC	Interrumpe la edición.
	F	Para conmutar si una entrada se ha de colocar en "frozen", es decir, ha de ser congelada. Las entradas congeladas se representan en blanco.
	ESC	Vuelve al menú principal.

Edit Memory Contents

Esta opción sirve para editar zonas de memoria mayores. Esto es especialmente útil para juegos de rol y de aventuras, en los que se han de cambiar nombre, descripciones, etc. Se soportan las siguientes teclas:

Tecla	Significado	
E	Pasa al modo de edición. En él se pueden modificar los contenidos de memoria mostrados actualmente. Hay definidas las siguientes teclas:	
	Tab	Conmuta entre el modo ASCII y el hexadecimal.
	Ctrl+S	Guarda los cambios.
	ESC	Finaliza la edición.

Tecla	Significado
G	Permite la introducción directa de un dirección de memoria, a la que debe saltar el editor.
H	Permite la conversión de un número decimal a un número hexadecimal y al revés.
N	Repite una búsqueda iniciada.
S	Busca una combinación ASCII o de números. La búsqueda se inicia en la posición de memoria actual.
Ctrl + G	Mediante esta combinación puede saltar directamente a una posición determinada en la memoria. Esto puede llegar a ser más rápido que ir a través de la *Table of Memory Locations*.
ESC	Mediante ESC vuelve de nuevo al menú principal.

Game Playing Speed

Game Wizard le ofrece la posibilidad de modificar la velocidad de ejecución de un juego. Esto puede tener sus ventajas para juegos de acción, y para aquellos que fueron escritos para los lentos 286. Puede ajustar el índice de velocidad con las teclas del cursor.

View Current Program Screen

Esta es una función muy útil. Muestra la pantalla actual del programa. Esto es muy útil cuando acaba de olvidar, por ejemplo, el valor exacto de una variable.

Crash to DOS

Esta función termina el programa actualmente en ejecución. Esto puede ser muy útil cuando el programa se ha vuelto a quedar "colgado", y no tiene ganas de reiniciar de nuevo el ordenador.

Estas han sido las funciones principales de Game Wizard. En este programa realmente tiene sentido registrarse, ya que la versión completa ofrece un conjunto de funciones muy ampliado, además de otros métodos de búsqueda. Si no tiene ganas de escribir un entrenador para un juego, este programa realmente es la mejor opción.

16.4 Componga con Scream Tracker 3.01b

El Scream Tracker 3.01 beta de Sami Tammiletho ("Psi" de Future Crew) se publicó a principios de 1994 y pertenece a los mejores "trackers" de música del PC. Su particularidad es que aparte del formato estándar MOD también es capaz de leer

y editar el formato S3M desarrollado por Future Crew. Con este formato S3M el Scream Tracker puede emplear 16 voces digitales (16 muestras simultáneamente) y 9 voces FM (AdLib). La siguiente descripción de las funciones del Scream Tracker es sólo una pequeña parte de las muchas funciones, y quiere servir como introducción. Como abreviación para Scream Tracker 3.01 beta emplearemos aquí ST3.

El manejo del Scream Tracker

Iniciar el Scream Tracker

Después de introducir *ST3* en la línea de comandos del DOS, se ejecuta Scream Tracker con una imagen de presentación, y la autodetección de la tarjeta de sonido. Si ST3 no encuentra la tarjeta de sonido, ha de indicar los parámetros a mano:

-s1 o -s5	SB
-s2	GUS
-mxxx	Mixing speed
-axxx	Dirección
-ixx	Interrupción
-cx	DMA de la tarjeta de sonido

```
Ejemplo: st3 -s1 -m210 -a220 -i7 -c1
```

Para un correcto funcionamiento de ST3 también es necesaria la presencia de suficiente memoria EMS. Si no es así, habrá de ejecutar un administrador de memoria adecuado como EMM386 en el CONFIG.SYS.

1. El menú principal (<Esc>)

Después de arrancar se encontrará directamente en el editor de patrones, pero de ello hablaremos más tarde. Con la tecla [ESC] puede ir al menú principal del Scream Tracker desde cualquier punto del programa. En el menú principal puede cargar y guardar MOD y S3M, llamar el Setup (Settings), reproducir o editar S3M/MOD, o comprobar la memoria disponible (Status). Muchas de estas funciones también se pueden alcanzar cómoda y rápidamente mediante las teclas de función.

2. La carpeta de temas (<F1>)

A la izquierda de la carpeta de temas se determina el orden en el que se han de reproducir los patrones (inglés pattern = una especie de tiempo en el que hay almacenados notas y otras informaciones musicales). Así por ejemplo se pueden repetir patrones, reproduciendo por ejemplo el patrón *00* en la posición *1*y más tarde, en la posición *10*, de nuevo el patrón *00*. Con la tecla [Tab] se puede conmutar al centro de la pantalla, en el que se determina la posición estereofónica de los distintos canales. A la derecha se pueden modificar la velocidad y el volumen del tema en el Song-Setup.

3. El editor de patrones (<F2>)

En el Pattern-Editor se puede crear un patrón. Un patrón tiene más o menos el siguiente aspecto. En horizontal se pueden reconocer cinco pistas (*01-05*), al lado se indica si se trata del canal derecho o izquierdo de la tarjeta de sonido (por ejemplo *L1* -> izquierda). Esto naturalmente sólo tiene sentido en tarjetas de sonido estereofónicas. Con la tecla [Tab] se puede conmutar entre las diferentes pistas. Así también se pueden alcanzar las pistas posteriores (*6-32*). En vertical, un patrón mide 64 líneas, es decir que se pueden introducir 64 notas por pista. Una pista a su vez se compone de los siguientes elementos:

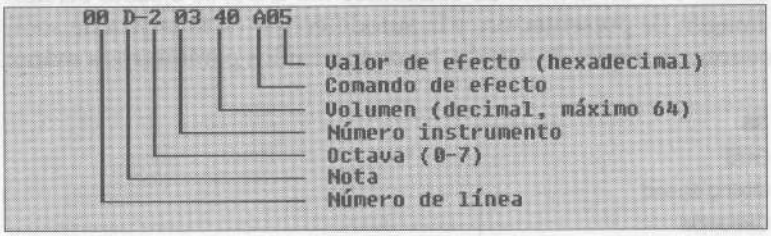

Figura 37: Estructura de una pista MOD

La posición del cursor de entrada en la pista se puede modificar mediante las teclas del cursor. La octava se puede aumentar o disminuir mediante * y / en el teclado numérico. Más adelante explicamos algunos efectos importantes, aunque aquí puede ser muy útil la página de ayuda ([F10]), que muestra todos los efectos. El punto [.] borra una nota. Las notas se introducen mediante el teclado, que emula el teclado de un piano. Otras teclas importantes del editor son [+] y [-], que sirven para cambiar el patrón, así como [Barra Espacio] para cambiar de instrumento.

4. El editor de instrumentos (<F3>)

Aquí se puede modificar la lista de instrumentos, cargando nuevos instrumentos, asignándoles nuevos nombres y volúmenes o reproducirlos como comprobación.

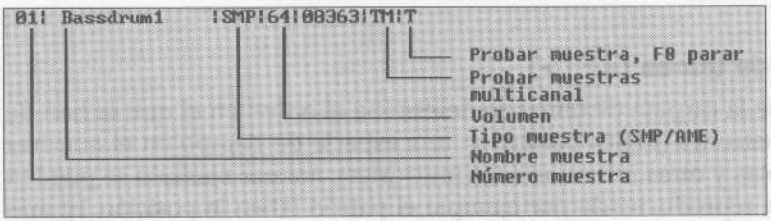

Figura 38: Un elemento de la lista de instrumentos

En esta lista se pueden encontrar tanto muestras (N. del T.: un "sample" es una porción de sonido digitalizado, SMP) así como los llamados sonidos FM de la tarjeta AdLib, que estarán marcados como AME en el tipo de sample. Se ha de tener en cuenta que los sonidos AdLib no se pueden reproducir con la Gravis Ultrasound.

Se puede cargar un nuevo sample colocando en cursor sobre el sample a sustituir o sobre un número de sample vacío y pulsando [Return] o [F4]. Así se accede directamente a la biblioteca de muestras, de la que se puede elegir un sample nuevo mediante [Return]. En el menú de la derecha se puede determinar el Loop de un sample, es decir, el tiempo a partir del cual se repite indefinidamente.

5. La biblioteca de muestras (<F4>)

Aquí tiene sentido el crearse varios directorios (en la línea de comandos, ST3\I), en los que se guardan las diferentes muestras. Puede darle al directorio el nombre que corresponde a las muestras que contiene. Por ejemplo: ST3\I\DRUMS para las muestras de batería.

6. Reproducir con el Scream Tracker

Reproducir tema (<F5>)

Se reproduce el tema actual con la secuencia de patrones determinada en la carpeta de temas. Con las teclas [RePág] y [Av Pág] se pueden mostrar los diferentes patrones en marcha y el espectrómetro. Se reproduce indefinidamente, hasta que se pulsa Stop ([F8]).

Reproducir patrón actual (<F6>)

Como [F5], pero mediante [+] y [-] se reproduce el patrón seleccionado.

Reproducir patrón actual, en posición actual (<F7>)

Como [F6], pero reproduce desde la posición actual del cursor.

Song stop (<F8>)

Detiene un tema, patrón o muestras repetidos.

7. Mostrar memoria (<F9>)

Esta función muestra la memoria libre e informaciones sobre el tema actual.

8. Página de ayuda (<F10>)

Esta tecla llama a la función de ayuda. Frecuentemente está diseñada específicamente para la parte del programa en la que se encuentra actualmente.

La composición de un S3M con el Scream Tracker

Instrumentos

El primer paso para componer un módulo es crear un sample adecuado. Para ello se pueden tomar prestados muestras de S3M ya existentes, incluso hay algunos incluidos con el Scream Tracker, o se pueden muestrear sonidos propios, por ejemplo con el VOC386 o DIGIPLAY 3.0, de CD de Audio o -lo más preferible- de teclados y sintetizadores. La ventaja del ST3 está en que puede procesar las muestras tanto en formato Amiga como en formato PC. El obtener sonidos AdLib ya es un poco más complicado, ya que no se pueden muestrear, sino que se obtienen especificando determinados parámetros. Si no se quiere dedicar mucho a ello, lo mejor es tomar los del FC Starport Intro, que también se encuentra en el directorio ST3.

Patrón

Ahora se puede intentar insertar en el editor de patrones, en diferentes pistas, voces de batería, bajo o melodía a nuestro entero gusto. Se ha de tener en cuenta que se han de mezclar los volúmenes de las diferentes voces en una relación adecuada y equilibrada.

Efectos

La gran cantidad de efectos a veces confunden al principiante, que comienza a componer sus primeros MOD o S3M. Ya que el manual inglés y la página de ayuda (F10) del editor de patrones describen extensamente todos los efectos, aquí nos limitaremos a algunos importantes que comuniquen el principio básico de cómo emplear efectos. El ST3 lamentablemente no emplea los comandos estándar del Amiga-ProTracker, por ello se ha de aprender de nuevo, si anteriormente se ha trabajado con otros trackers. Aquí unos cuantos ejemplos de efectos importantes y su significado:

Axx Velocidad

El valor *xx* indica la velocidad del tema en hexadecimal. Se puede modificar, por ejemplo, con cada patrón. *01* es la velocidad más rápida posible, y *FF* es la más lenta. Por defecto siempre se encuentra ajustado el valor *06*.

D0x reducir rápidamente el volumen (slide down)

El valor *x* indica con qué velocidad se reduce el volumen. Una *x* minúscula decrementa el volumen lentamente, una *X* mayúscula rápidamente. Con este efecto se puede obtener un bonito efecto de "fading", si después de iniciar la nota se

colocan varios de estos comandos en secuencia, y se mantiene bajo el valor de *x*. La velocidad con que el tono se decrementa también depende de la velocidad general del patrón.

Dx0 aumentar rápidamente el volumen (slide up)
Como antes, pero con el límite superior de volumen *64*.

Gxx Portamento del tono
El valor *xx* indica la velocidad con la que se modifica el tono de la nota anterior, para alcanzar la nota actual (la del comando *Gxx*).

También es muy útil en el manejo de efectos, el escuchar muchos temas de ejemplo, y aprender de ellos. Hay muchos trucos y secretos que se pueden oír en ellos. Y para la composición se aplica lo mismo que para la programación: Learning by doing! De modo que no se deje impresionar por la gran cantidad de funciones, sino que simplemente comience a componer.

16.5 Muestrear con VOC386

El programa VOC368 de Christoph Vaessen es probablemente uno de los mejores programas de sampling que actualmente hay disponibles para PC. El programa es shareware, y la tarifa de registro es de sólo 30 marcos alemanes (Sólo está disponible la versión alemana). En cualquier caso es una inversión adecuada, ya que no sólo apoya al autor, sino que obtiene una versión optimizada en velocidad y requerimiento de memoria.

El programa necesita al menos un procesador 386, y un 486 aumenta la velocidad de trabajo. VOC386 se puede manejar con el teclado, pero el ratón es muy aconsejable. El programa funciona conjuntamente con todos los administradores de memoria corrientes, simplemente debería procurar que disponga de mucha memoria ya que sino se producirá el volcado a disco.

Si tiene problemas al ejecutar VOC386, eventualmente ha de modificar una entrada en el archivo VOC386.INI. Puede encontrar informaciones más extensas sobre ello en el archivo VOC386.DOC, que contiene un buen manual en castellano.

Las funciones básicas de VOC386

A continuación les presentaremos las funciones básicas de VOC386. Si necesita informaciones más extensas, puede encontrar la documentación en el CD adjunto.

Cargar muestras

Vaya al punto de menú de la esquina superior izquierda del menú y pulse sobre la entrada *Laden*. Aparece una máscara de selección de archivos, por ejemplo *.*voc*. Esta se puede modificar o aceptar directamente con [Return]. Entrará en un menú de selección de archivos, en el que puede elegir el deseado.

Guardar muestras

El punto de menú para guardar muestras se encuentra una entrada por debajo de la de cargar. Vaya con el ratón sobre ella, y pulse con el botón del ratón. Emplee el botón izquierdo del ratón, para guardar el tramo seleccionado en rojo, y el botón derecho del ratón para guardar todos los datos de sampling que se encuentran en la memoria.

La digitalización

Mediante el punto de menú *Digitalisieren* se inicia la grabación de un sample a través de la tarjeta Sound Blaster. Puede terminar la grabación mediante la tecla [Ctrl]. Cualquier dato que se encuentre en memoria será sobrescrito sin piedad.

Sekunden/Byte (Segundos/byte)

Con ello puede determinar si las informaciones sobre el sample se han de mostrar en segundos o en bytes (longitud real del sample). Si pulsa sobre este punto de menú con el botón izquierdo del ratón, la indicación se realizará en segundos, mientas que se lo selecciona con el botón derecho del ratón, se hará en bytes.

Ausschneiden (Recortar)

Esta función copia la zona seleccionada en un búfer. Si selecciona este punto con el botón izquierdo del ratón, se copia la zona marcada en rojo al buffer. En una llamada a través del botón derecho del ratón se copia la zona seleccionada al búfer y simultáneamente se recorta del sample.

Einsetzen(Insertar)

La llamada de esta función con el botón izquierdo del ratón provoca la inserción del buffer en la posición seleccionada. Si se llama con el botón derecho del ratón se sobrescriben los datos del sample.

Ersetzen (Sustituir)

Esta función sobrescribe la zona seleccionada con los datos del buffer. No importa qué botón del ratón emplee.

Abspielen (Tocar, desarrollar)

Si pulsa sobre este punto de menú, escuchará el sample que se encuentra en memoria. La reproducción se inicia inmediatamente y se puede interrumpir en cualquier momento mediante la tecla Ctrl.

Buffer abspielen (Tocar búfer)

Mediante esta función se pueden escuchar los datos de sampling que se encuentran en el buffer.

+ Speed/- Speed (+ velocidad/- velocidad)

Para aumentar o reducir la frecuencia de sampling, puede utilizar estos dos puntos de menú. El botón izquierdo del ratón modifica la frecuencia en 100 Hz, el botón derecho en 1.000 Hz.

Repeat ein/aus (Repetir)

Con esta opción puede activar y desactivar el modo *Repeat*. Pulse con el botón izquierdo del ratón para activarlo, o con el derecho para desactivarlo. En el modo *Repeat* se vuelve a reproducir un sample una vez que se ha llegado al final del mismo, hasta que se pulsa la tecla Ctrl. El ajuste actualmente activo se muestra en azul.

Löschen (Borrar)

Active *Löschen* para rellenar una zona de un sample con un período de silencio. Si ejecuta el comando con la tecla izquierda del ratón, se borra la zona seleccionada actualmente, con una pulsación sobre el botón derecho del ratón se borra el sample completo.

Zoom in/Zoom out

Para ver la zona seleccionada en una resolución mayor, seleccione el punto de menú *Zoom in*. Para volver atrás en el nivel de zoom, hay dos posibilidades: Mediante el botón izquierdo del ratón vuelve al nivel anterior de zoom y mediante el botón derecho volverá a tener en pantalla el sample completo.

Lauter/Leiser (Más alto/más bajo)

Estas dos funciones sirven para aumentar o reducir el volumen del sample. El botón izquierdo del ratón vuelve a modificar la zona seleccionada, y el derecho el sample completo.

Schneller/Langsamer (Más rápido/más lento)

Estos dos puntos de menú sirven para aumentar o reducir la velocidad de reproducción. El botón izquierdo modifica la zona seleccionada y el derecho el sample completo.

Hochpass/Tiefpass (Paso alto/paso bajo)

Estas funciones sirven para resaltar tonos especialmente altos o bajos. Se le pedirá que introduzca un factor *1* significa que sólo se modifican tonos muy altos o bajos, un *9* permite también la alteración de tonos medios. Para los botones del ratón se aplica lo anteriormente dicho.

Einblenden/Ausblenden (Ampliar/extinguir)

En el caso de *Einblenden* se incremente el volumen desde 0 hasta el volumen activo, con *Ausblenden* se reduce del actual hasta 0. Seleccione el botón izquierdo del ratón para modificar la zona seleccionada, y el botón derecho para cambiar todo el sample.

Los diferentes efectos de VOC386

VOC386 pone a su disposición varios efectos para distorsionar el sonido. Para ellos se aplican las funciones habituales de los botones del ratón.

El efecto *Hall* sirve para generar un sonido espacial, como de una gran iglesia. El efecto *Echo* genera el eco tan conocido de las montañas. Mediante la función *Rückwärts (hacia atrás)* puede reproducir a la inversa una zona o todo el sample. El *Rauschfilter (filtro de ruido)* sirve finalmente para colocar en silencio total los períodos de silencio que tienen algo de ruido de fondo. De nuevo *1* indica un tolerancia baja, y *9* la máxima. Estas han sido todas las funciones de VOC386. Ahora debería estar en disposición de emplear el programa con plena satisfacción. Las respuestas a preguntas más profundas se encuentran en el estupendo manual en castellano.

16.6 Assembler A86

Para la instalación copie A86V370.ZIP a un directorio, y descomprima el programa. El ensamblador shareware A86 representa una alternativa a los ensambladores comerciales MASM y TASM. Las características a destacar de este producto son las

estructuras sencillas y muy próximas al procesador, que simplifican el trabajo en el caso de proyectos pequeños. Los textos fuente simplificados naturalmente conllevan una pequeña incompatibilidad, que lleva a que muchos programas TASM han de ser adaptados antes de ser compilados con A86. Por ejemplo, se ignoran algunas importantes instrucciones de ensamblador del TASM/MASM como ASSUME, de modo que uno mismo se ha de preocupar de los Segment-Overrides (*mov ax, CS:var*). El A86 también ofrece algunas mejoras que simplifican determinadas secuencias que aparecen con frecuencia:

- Un PUSH de varios registros se puede realizar con un sólo comando:

```
PUSH ax, bx, cx en vez de  PUSH ax; PUSH bx; PUSH cx
```

- Se puede acceder directamente a los registros de segmento, el ensamblador por ejemplo permite la instrucción:

```
mov ds, cs
```

- Las instrucciones INC y DEC obtienen otro operador adicional, que indica el sumando a añadir (*INC ax,2 = incrementar AX en 2*).
- Las instrucciones condicionales se han simplificado, en vez de

```
  jnz label
  mov ax, bx
label:
```

- ahora se puede escribir:

```
if z mov ax, bx
```

En la documentación puede encontrar algunas ampliaciones de poca utilidad, en el capítulo 5 (A05.DOC). Naturalmente, el procesador sigue sin poder trabajar con estas instrucciones, sino que durante el ensamblado se traducen en la forma ya conocida, más larga. Una ampliación muy práctica se realizó en el tratamiento de errores: Los mensajes de error no se visualizan en pantalla, sino que se incluyen directamente en el texto fuente. Durante el siguiente ensamblado se eliminan automáticamente. Esta facilidad sin embargo ya no tiene tanta importancia hoy en día, ya que los ensambladores modernos se encuentran integrados en los entornos existentes de Pascal y C, con lo que los mensajes de error se listan de una forma aún más cómoda en las ventanas de mensaje. El A86 se llama con

```
fA86 archivofuente [to] [archivobinario] [archivosimbolos] [Opciones]
```

Los parámetros en corchetes son opcionales. El archivo fuente es el archivo con el texto fuente. El siguiente *to* es opcional, y muestra que el resultado del ensambla-

do irá a parar al *archivo-binario*. En base a la extensión se decide si será un archivo COM (.COM), un archivo objeto para linkar con otros archivos (.OBJ) o un archivo binario para programar EPROMS (.BIN).

Se escribe un listado de los símbolos empleados en el programa al archivo simbólico, que ha de llevar la extensión .SYM. Si se especificaron varios archivos fuente, se ordenan sus nombres alfabéticamente y se ensamblan en este orden, como si de un gran archivo se tratara. Se pueden añadir algunas opciones, que, entre otras cosas, intentan aumentar la compatibilidad con TASM/MASM:

+D:

El A86 normalmente distingue la base numérica en función de la primera cifra. Si es *0*, se trata de un número hexadecimal, de lo contrario, de un número decimal. Mediante la opción *+D* estos números -de forma compatible a TASM/MASM- se considerarán siempre como números decimales, a menos que se declaren de otra forma añadiendo *h*.

+G15:

Aquí se desactivan algunas particularidades durante la generación de código, de modo que el código objeto sea compatible TASM/MASM. Los detalles puede verlos en la documentación (A03.DOC).

+O:

Es idéntico a añadir un archivo objeto en la línea de comando: en vez de un archivo COM se generará un archivo objeto.

+f:

Corresponde al parámetro TASM */E* y activa la emulación de coprocesador (sólo al linkar con un lenguaje de alto nivel). El porteado de programas terminados, sobre todo largos, al A86 requiere algún esfuerzo, pero es posible. Primero se debería ajustar una compatibilidad lo más amplia posible mediante las opciones *+G15* y *+D*. Sin embargo, aún así se habrán de hacer algunos cambios en el texto fuente.

Como el A86 no posee la potente directiva de compilador ASSUME (por convicción, como indica el autor. Cita: "[...] the ASSUME mechanism creates far, far more confusion that it solves. So I scrapped it; ...», archivo A06A.DOC), se han de proveer todos los accesos a segmentos que no corresponden al segmento estándar, con el Segment-Override manual correspondiente. Si la variable *Var* por ejemplo se encuentra en el segmento de código, se ha de escribir lo siguiente para leerla: *mov ax, CS:Var*, en vez del sencillo *mov ax, Var*.

En el A86 se consideran las etiquetas de la forma *L14*, es decir, una letra seguida de una o varias cifras, como etiquetas locales, que pueden aparecer más de una vez en el texto fuente. Esto tiene sin embargo la desventaja de que se ha de tomar una decisión inequívoca de qué etiqueta es la que indica en un salto. El A86 emplea para ello el carácter », que indica un salto hacia adelante. *jmp »L2* salta a la siguiente etiqueta *L2*, mientras que *jmp L2* va a la anterior. Por ello ha de incluir los signos de "mayor que" para los saltos hacia adelante en su texto fuente TASM/MASM.

Las macros se han de someter a grandes modificaciones. A86 no conoce nombres para los parámetros, en vez de ello se han de emplear las variables *#1* hasta *#9*. El final de la macro se indica con *#em*. En el archivo A12.DOC se listan otros sinónimos para los comandos de macro. Aquí también se encuentra una instrucción para portear textos fuente en ambas direcciones.

16.7 Debugger D86

Para la instalación copie D86V370.ARJ a un directorio, y descomprima el archivo. Como complemento para el ensamblador A86 se emplea el debugger (depurador) D86 del mismo autor. Con este, de forma parecida al Turbo Debugger de Borland, se pueden ver archivos ejecutables, y procesarlos paso a paso. Al contrario que en la primera generación de depuradores (como DEBUG.COM de MS-DOS) se muestran constantemente informaciones importantes en la pantalla. Esto afecta tanto al código del programa en la memoria, los registros, las banderas así como las zonas de memoria seleccionadas.

El D86 se apoya estrechamente en el A86, incluso se mantiene a éste simultáneamente en memoria. Con ello no sólo es posible añadir nuevas instrucciones de ensamblador al código existente, sino también enviar instrucciones directamente al procesador, como también se puede hacer en los intérpretes de Basic. Por ejemplo, no existe ninguna posibilidad de modificar el contenido de un registro desde el depurador, esto se realiza ejecutando la instrucción *Mov* correspondiente.

También es posible la depuración simbólica, es decir, el empleo de etiquetas e identificadores en vez de las direcciones tan poco elocuentes, siempre que exista un archivo SYM en el que se encuentren todos los símbolos que se emplean en el programa. Este archivo SYM se genera ensamblando el texto fuente con el A86 o mediante la conversión de los archivo MAP de otros ensambladores. La última posibilidad sólo se ofrece, sin embargo, a los usuarios registrados. El D86 se ejecuta mediante:

```
D86 [Opciones] [Nombre [Parámetros ] ]
```

Los parámetros en corchetes se pueden omitir. El programa especificado por *Nombre* se carga al ejecutar el depurador, y se le pasan los parámetros. La llamada de

D86 XCOPY A: por ejemplo le deja analizar cómo copia archivos el comando *Xcopy* desde la unidad A:. Sólo existen dos opciones, de las cuales una hoy en día es superflua, ya que procura un soporte de ordenador no compatibles IBM. La segunda opción *+v* activa el soporte de dos monitores (un monitor en color y otro monocromo) durante la depuración.

Después de la ejecución del programa se puede ver arriba a la izquierda la ventana de desensamblado. Aquí se representa el código de programa de la posición actual. Abajo a la izquierda se puede ver una lista de los registros y las banderas. Las últimas se muestran si están activas, y se borran cuando están a cero. Las denominaciones en general corresponden a la norma, simplemente la bandera de paridad (*p*) se simboliza aquí por una *e*.

Arriba a la izquierda se puede ver una ventana multiuso. Aquí se pueden consultar textos de ayuda, zonas de memoria y mensajes de estado. Abajo a la derecha se muestran en seis líneas zonas de memoria determinadas. La línea inferior, al inicio del programa inicializada con *0:*, muestra el stack (la pila). Según la cantidad de palabras que haya en el stack, su contenido se va mostrando secuencialmente.

Pulsando la combinación de teclas Alt + F10 se conmuta la ventana multiuso entre modo de ayuda y de estado. En estos modos se realiza la selección de la información a mostrar mediante la tecla F10. En el modo de ayuda se encuentran las indicaciones de las teclas de función, de las teclas sencillas y de las combinaciones de teclas con Ctrl. En el modo de estado se conmuta entre mensaje de Copyright, informaciones de estado, extracto de memoria y coprocesador. Las informaciones de estado también se pueden llamar directamente mediante combinaciones de teclas:

Combinación de teclas	Información de estado
Ctrl + F	Coprocesador
Ctrl + N	Página de memoria siguiente
Ctrl + P	Página de memoria anterior
Ctrl + S	Informaciones de estado
Ctrl + T	Borra el stack
Ctrl + D	Un byte adelante en el código del programa
Ctrl + U	Un byte atrás en el código de programa
Ctrl + E	Salto al final del programa
Inicio	Salto a la última instrucción ejecutada, después al inicio del programa

El extracto de memoria en la ventana arriba a la izquierda siempre representa una extensión del último bloque de memoria seleccionado en las líneas de memoria (1 a 6) abajo a la derecha. Mediante las teclas de función se puede influir en la ejecución del programa a analizar:

Teclas	Significado
F1	Paso único en funciones, pero no en interrupciones
Ctrl + X	Paso único incluso en llamadas a interrupción
F2	Paso único por encima de llamada a procedimiento
F3	Repetir última instrucción introducida
F6	Ejecución hasta el siguiente RET
F7	Introducir/ensamblar nuevas instrucciones
F9	Ejecuta el programa desde la última posición hasta la actual
Shift + F7	Marca posición para búsqueda posterior (véase tecla F)
Alt + F9	Dibujar pantalla de nuevo

La ejecución directa de instrucciones en ensamblador se realiza mediante la simple introducción de las mismas. Si se pulsa por ejemplo la tecla M, la ventana de ayuda (suponiendo que esté activada) avisa de que se debe continuar con la introducción de la instrucción, es decir, que se complete la instrucción *OV AX,BX*. El depuradores controlado mediante secuencias de instrucciones semejantes a las directas de ensamblador, y principalmente se emplean las siguientes funciones:

B, Dirección [,Dirección]

Breakpoint activa uno de los dos Breakpoints disponibles, o ambos a la vez, para interrumpir la ejecución del programa en este punto.

F, Longitud [,Offset]

Find busca el tramo de código marcado con Shift + F7, a partir de la posición actual, y se comparan Longitud bytes para desplazar el código de programa en Offset bytes hacia abajo en caso de una búsqueda con éxito.

G [,Dirección] [,Dirección]

Go ejecuta el programa y fija hasta dos Breakpoints temporales, que se vuelven a borrar después de la ejecución del programa.

J, Dirección

Jump salta a la dirección indicada.

L [,Dirección] [,Nombre archivo]

List genera un listado de la posición actual hasta Dirección en la impresora o en un archivo. Si no se especifica ninguna dirección, se imprime el programa completo.

O, límite inferior [,límite superior]

Detiene un programa cuando se llama la función número *límite inferior* de la interrupción DOS 21h. Si también se especifica un *límite* superior, se vigilan todas las funciones del rango especificado.

Q

Quit termina el debugger.

W

Write escribe un programa al disco duro (sólo para archivos COM) y crea de nuevo la tabla de símbolos (*.SYM).

Sólo faltan las líneas con los volcados de memoria abajo a la derecha. Estas se seleccionan pulsando la tecla numérica adecuada. Ahora se puede definir qué rango se ha de mostrar de esta forma, donde se ha de emplear la siguiente sintaxis:

Tipo, Segmento, Offset

Segmento y Offset indican la posición del extracto de memoria, donde también se pueden emplear registros. En la especificación del tipo se han previsto muchas posibilidades del control, aquí sólo presentaremos las bases:

Tipo	Significado
A	ASCII, Cadenas de caracteres con signos de control
B	Bytes hexadecimal
D	Palabras decimales
F	Coma flotante (Coprocesador)
N	Bytes decimales
S	Cadenas ASCIIZ (correspondientes a C)
W	Palabras hexadecimales

16.8 TOS-Copy, un práctico programa de copia

Para la instalación, copie TOS-COPY.EXE a un directorio, y ejecútelo. El programa se desempaqueta automáticamente. TOS-Copy es un programa de copia para disquetes, que tiene algunas particularidades con respecto a DISKCOPY de MS-DOS. TOS-Copy puede leer un disquete en una pasada, aprovechando memoria XMS y disco duro como buffer. Además, los disquetes que se han leído una vez, se podrán copiar múltiples veces. TOS-Copy se llama con

```
TC [Fuente] [Destino]
```

La indicación de unidad fuente y destino es opcional. Ambos también se pueden elegir en el programa. Después del inicio del programa se puede encontrar arriba a la derecha un indicador de estado, que indica las unidades de disquete existen-

tes, la memoria XMS libre y durante la copia el tiempo estimado que durará ésta. Arriba a la izquierda se encuentra una ventana multiuso, que muestra cosas diferentes según la situación y que además contiene un menú. Debajo se encuentra un indicador de estado del proceso de copia actual con línea de texto e indicador de pista.

La ventana multiuso posee tres niveles, que se pueden cambiar mediante el botón *OK* (un nivel arriba) y *SETUP* (un nivel abajo). El nivel inferior posee algunas opciones de sonido, el siguiente sirve para indicar las unidades fuente y destino (la flecha intermitente muestra la combinación actual). Aquí también se pueden realizar copias entre diferentes tipos de disquetera. (3,5" a 5,25" y al revés). En realidad todas las combinaciones son posibles, pero no recomendables. Una copia de 1,44 MBytes (3,5") a 1,2 MByte (5,25") generará en los menos de los casos un copia libre de errores. El nivel superior muestra una ampliación de las unidades fuente y destino y pone a disposición los conmutadores de copia *READ, WRITE* y *COPY*. *READ* lee un disquete de la unidad fuente al buffer, *WRITE* escribe este buffer a un disquete, incluso múltiples veces si se desea, lo que produce una serie de copias idénticas.

COPY reúne ambos procesos. Después de leer el disquete fuente, se le pide al usuario que inserte el disquete de destino, que será escrito. Después de la llamada de *COPY* el contenido del disquete sigue encontrándose en el buffer, y puede ser escrito a más disquetes con *WRITE*. El último punto de menú *QUIT* termina el programa.

16.9 VPIC 6.1 - Visualizador gráfico universal

Para la instalación, copie VPIC61.EXE a un directorio y ejecútelo. El programa se desempaqueta automáticamente. VPIC es un clásico entre los conversores y visualizadores gráficos. Se distingue sobre todo por su gran selección de modos gráficos, que se pueden definir individualmente para cada juego de chips gráficos.

El responsable de esta adaptación es el programa CONFIG.EXE, con cuya ayuda se puede elegir un controlador gráfico adecuado de la extensa lista. Si a pesar de ello no hubiera ningún controlador que trabaje con su tarjeta gráfica (muy poco probable), puede crear fácilmente su propio controlador (instrucciones en CONFIG.DOC). VPIC se llama mediante

```
VPIC [Nombre de imagen] [Opciones]
```

Si no se especifica ningún nombre de imagen, se pueden elegir en el menú de archivos. Las opciones son muy extensas, pero principalmente sirven para auto-

matizar procesos, para generar presentaciones y por ello aquí no son de interés. En el menú se pueden seleccionar los archivos mediante la tecla [Barra Espacio] y se pueden mostrar mediante [Return]. La imágenes visualizadas se pueden convertir a otros formatos mediante pulsación de diferentes teclas. Estas son:

Tecla	Formato/Función
[C]	Dr. Halo (.CUT)
[D]	Deluxe Paint (.LBM)
[G]	GIF
[P]	PIC
[R]	ColorRIX (.RIX)
[S]	TIF
[T]	TGA
[W]	Windows-BMP
[Z]	PCX
[A]	Rotación de la paleta (para fractales)
[B]	Creación de una imagen en B/N
[F1] + [F10]	Corrección de los colores básicos
[Alt] + [F10]	Restaurar la paleta original
[/] o [?]	Ayuda, visualizar las teclas disponibles

En el menú de selección hay disponibles otras teclas adicionales aparte de [Return] y [Barra Espacio]:

Tecla	Significado
[F1]	Ayuda
[F2]	Iniciar una presentación automática: las imágenes seleccionadas se muestran una detrás de otra. El retardo se ajusta con [Alt] + [D]
[F3]	Informaciones sobre el archivo seleccionado
[F4]	Conmutar 16/256 colores
[F5] / [F6]	Aumentar/Disminuir resolución
[F7]	Fijar resolución (LOCK, Automatismo desconectado)
[Alt] + [F7]	Fijar resolución en HiColor
[F9] / [Alt] + [F9]	Ajustar vía de acceso
[Alt] + [C]	Copiar los archivos seleccionados
[Alt] + [M]	Mover los archivos seleccionados
[Alt] + [X]	Borrar los archivos seleccionados

16.10 PV 2.41 - Mostrar y convertir gráficos

Para la instalación copie PV241.EXE en un directorio y desempaquete el programa. PV en un visualizador y conversor gráfico con algunas particularidades en el

ámbito de procesamiento de imágenes. Posee diferentes filtros y métodos de dithering (tramado), con los que se pueden crear efectos interesantes. PV puede leer casi todos los formatos gráficos existentes, comenzando por los formatos comunes GIF, PCX, etc. pasando por las diferentes versiones TIF y TGA, hasta los formatos exóticos como FLM (generador de cartas de ajuste) y SAT (imágenes de satélite meteorológico). Al escribir, sin embargo, el programa se concentra en los formatos más comunes. La llamada se realiza mediante:

```
PV [Nombre de imagen] [Vía de destino] [Opciones]
```

La mayoría de ajustes que se puede realizar desde el entorno de menús, ya se pueden activar en las opciones. Aquí sólo nombraremos las más importantes, añadiendo directamente el identificador de formato (por ejemplo /c1):

Parámetro	Significado
/c	conversión a los siguientes formatos: j JPEG hj JPEG de alta compresión 0 IMG 1 TGA 2 LBM h2 Amiga-LBM 3 CUT 4 GIF 5 TIFF h5 Motorola-TIFF 6 SCx 7 PCX 8 PIC 9 BMP h9 RLE
/m	representación/conversión monocroma
/n	no borrar la pantalla al finalizar, sino seguir mostrando la imagen. Apto para incluirlo en programas propios.

Si no se especifica ningún nombre de archivo, o si el archivo no se puede abrir, se entra en el menú *Datei* (Archivo). Aquí puede encontrar a la izquierda la lista de los archivos del directorio actual, por los que puede desplazarse con las teclas del cursor. Con [Return] se representa el archivo seleccionado. Puede cambiar de unidad de disco mediante [L]. A la derecha se encuentra una visualización de los ajustes actuales (modo gráfico, máscara de archivos, etc.) con las teclas correspondientes. Puede obtener una explicación extensa de las teclas mediante [F1]. Aquí queremos dar una breve explicación de las más importantes:

Tecla	Significado
O	Conmuta entre el modo de visualización y de conversión. En el modo de conversión se pueden convertir archivos, sin mostrarlos.
X	Activa un modo de vídeo especial (predefinido como VESA-VGA auto).
M	Conmuta entre conversión/visualización monocroma y en color.
P	Indica el método con el que se deben calcular las informaciones de color. *RGB mit HiColor* activa la representación en TrueColor, *auf VGA reduziert* y *Farboptimierung* provocan la reducción a 256 colores.
U	Normalmente PV obtiene las informaciones como altura y profundidad de color de la cabecera del archivo, mediante U se pueden introducir de nuevo para cada imagen.
Inicio / Fin RePág / Av Pág F9 / F10	Modifican brillo, contraste y saturación de la imagen completa o de componentes de color individuales.
R, G, B, Barra Espacio	Seleccionar componentes de color individuales o de la imagen completa para la manipulación anterior.
I	Invierte la imagen.
M	Genera una imagen en B/N.
- + X, + + X	Aumentar/reducir en dirección X.
- + Y, + + Y	Aumentar/reducir en dirección Y.
U	Undo, restaurar la imagen.
D	Tramado en escala de grises (8 = 1 Bit, 1 = 256 grises).
F	Filtro, según el número de filtro se pueden activar aquí suavizadores, resaltadores, filtros de mosaico o similares.
T	Reflejar y rotar.
W	Determinar sección de imagen para operaciones.

Las (combinaciones de) teclas que se listan bajo la opción de línea de comandos */c*, también sirven durante la representación de imagen para convertir al formato correspondiente.

16.11 Convertir gráficos con Image Alchemy 1.7

Puede instalar el programa copiando ALCHM17.ZIP a un directorio y descomprimiéndolo allí. Para crear la documentación ALCHEMY.DOC, ha de ejecutar MANUAL.EXE. Image Alchemy es un potente conversor de gráficos para DOS. Aparte de las conversiones de múltiples formatos, se pueden modificar con facilidad las medidas, paletas y otros datos. Alchemy se llama mediante:

```
ALCHEMY archivo-fuente Opciones [archivo-destino]
```

Si no se especifica ningún formato de salida, en cualquier caso se ha de emplear uno de los siguientes parámetros, para obtener la visualización en pantalla. Todas las opciones *V* pueden llevar además la resolución horizontal.

-v	8 Bits (256 colores)
-V	8 Bits, adaptar imagen a la resolución
--v	Como *-v*, pero TrueColor
--V	Como *-V*, pero TrueColor

Las opciones más comunes son:

-x	Mostrar informaciones de la imagen
-h	Mostrar ayuda
-o	Sobrescribir archivo

El tamaño de la imagen se puede manipular con las siguientes opciones:

-X[t]n	Escalar a *n* píxeles en dirección x, emplear método *f*.
-Y[t]n	Como *-X*, pero en dirección y
-Dn m	Indicar resolución en dpi (n * m)

Se pueden modificar los colores mediante:

-n	Profundidad en Bits/Píxel (por ejemplo 16 para 16 Bits/Píxel)
-b	Crear B/N
-cn	Determinar el número de colores
-dn	Indica el método de tramado (dithering, 0-5, 20-22)
-f	Cargar paleta de otra imagen, práctico cuando varias imágenes han de tener la misma paleta
-N	Invertir imagen

Como formatos de salida se dispone de todos los formatos corrientes, incluso alguno tan extraño como PDS, que es empleado por la NASA para imágenes de satélite en CD. Se obtiene una lista de todos los formatos disponibles mediante la llamada de ALCHEMY -h2 y -h3.

16.12 Ver y convertir gráficos bajo Windows con PaintShop Pro

Puede instalar el programa, copiando PSPPRO.EXE a un directorio de su elección y ejecutarlo, se descomprime automáticamente. También bajo Windows hay una gran cantidad de conversores gráficos. Un ejemplo a destacar especialmente, es el Picture Shop Pro. Éste no sólo puede visualizar imágenes y convertirlas, sino que

también puede prestar sus servicios en la edición de las mismas. Posee extensas posibilidades de corrección de color y para filtrar. Como programa de Windows (lamentablemente en inglés) el control el muy intuitivo, de modo que podrá aprender con facilidad.

En el menú *File* se encuentra una opción muy interesante al lado de las funciones estandarizadas como *Open*, *Close* o *Print*, la opción *Batch*. Aquí puede indicar instrucciones masivas al conversor. En la parte izquierda del cuadro de diálogo ha de elegir la vía de acceso fuente, y seleccionar los archivos a convertir (si simultáneamente pulsa la tecla [Shift], se pueden marcar varios archivos). En la mitad derecha se han de seleccionar la vía de acceso de destino y el formato de salida. Después, PaintShop Pro convertirá todos los archivos seleccionados al formato especificado.

El menú *View* ajusta el factor de *Zoom*, y activa las herramientas y el histograma. La ventana de herramientas contiene iconos para el zoom (Lupa: botón izquierdo del ratón para aumentar, derecho para reducir), para desplazar el contenido de la imagen (mano), para seleccionar una zona (rectángulo discontinuo), para editar la selección (flechas) y para copiar el trozo a una nueva posición (dos rectángulos). El histograma representa un resumen de los diferentes brillos de la imagen. En el eje x se encuentran los brillos en orden ascendente. En el eje y se encuentran los valores porcentuales.

En el menú *Image* se puede reflejar la imagen de forma vertical (*Flip*) y horizontal (*Mirror*), así como rotarla. *Resample* y *Resize* ofrecen la posibilidad de escalar la imagen a nuevas medidas. *Resample* añade además un suavizado mediante Anti-Aliasing. Además se dispone de una gran cantidad de filtros. Los filtros *Edge* se emplean para detectar y reforzar los bordes. Los filtros *Normal* pueden suavizar (*Blur/Soften*) y resaltar (*Sharpen*). Los filtros *Special* son muy interesantes, con sus efectos especiales. Destacables son sobre todo *Despeckle* y *Emboss*. *Despeckle* elimina la "nieve" que aparece sobre todo al escanear imágenes, y mejora la calidad de estas imágenes enormemente. *Emboss* crea un efecto de relieve, que da una impresión tridimensional.

Bajo *Colors* se pueden manipular los colores de una imagen. *Brightness/Contrast* regulan el brillo y el contraste de la imagen completa. *Grey-Scale* crea un imagen en escala de grises y *Negative Image* invierte la misma. Mediante *Red/Green/Blue* se pueden regular las distintas componentes de forma individual. La paleta se puede modificar bajo *Edit Palette* (siempre que la imagen posea una paleta, la imágenes en High- y TrueColor no entran en esta categoría). Una información muy interesante la da el punto de menú *Count Colors Used*, que cuenta los colores realmente empleados en la imagen. Con ello se puede estimar la pérdida de calidad al reducir el número de colores. Se sorprenderá de qué pocos colores emplean incluso imágenes en TrueColor (habitualmente pocos miles, aunque teóricamente son posibles 16,7 millones).

Mediante *Decrease* e *Increase Color Depth* se pueden convertir las imágenes de una gama a otra de color. Por ejemplo de TrueColor a 256 colores. Una función que a menudo es realizada por programas externos ha sido integrada bajo el punto de menú *Capture*. Aquí se pueden realizar hardcopies (reproducciones) de la pantalla de Windows o de partes de ella. Si se ha elegido uno de los puntos de este menú, se provoca el proceso de captura en sí mediante la pulsación del botón derecho del ratón. Las diferentes posibilidades que se ofrecen se muestran en la siguiente tabla:

Función	Significado
Area	Permite elegir el área que quiere copiar con el ratón antes del proceso de captura.
Full Screen	Copia la pantalla completa.
Client Area	Copia el contenido de la ventana actual.
Window	Copia la ventana con marco y barra de título.

16.13 Graphic Workshop for Windows - Convertir y visualizar gráficos

Para la instalación del programa ha de copiar GWSWN11I.ZIP a un directorio y desempaquetarlo. Desde Windows ejecute el programa SETUP.EXE, e introduzca la vía de acceso de destino. El Graphic Workshop for Windows es otro representante de la clase de conversores gráficos para Windows. Quien conozca las versiones anteriores de este programa se sorprenderá durante la primera ejecución de los llamados "Thumbnails". Se trata de pequeños gráficos de resumen, que se crean para cada archivo gráfico. Gráficos aún no registrados han de ser añadidos primero mediante *Add Thumbnails* del menú *Thumbnails*. Quien valore más una gran cantidad de gráficos en vez de grandes iconos, puede desactivar esta función mediante *Use Thumbnails*.

Si se pulsa con el ratón sobre estos *Thumbnails* o los nombres de archivo, se seleccionan para un posterior procesamiento (incluso varios a la vez). Después se pueden someter los gráficos seleccionados a determinados procedimientos, bien mediante los botones o los puntos de menú del mismo nombre. *Convert* convierte los archivos seleccionados al formato de destino, que se determina en la siguiente selección.

Crop sirve para mostrar el gráfico completo que siempre se adaptará a la pantalla. Con el ratón se puede seleccionar una zona, que puede ser guardada con *Save*. *Next* pasa a la siguiente imagen, si hay varias seleccionadas, y *Get Info* muestra las informaciones de la imagen.

View es muy parecido a *Crop*. Siempre representa la imagen 1:1, pudiéndose aumentar con *Zoom*. *Save* guarda el gráfico completo, después de que haya podido ser modificado en sus colores, contraste o intensidad mediante *Adjust*.

Dither sirve para la reducción de la imagen seleccionada a blanco y negro, es decir, 1 bit. Se pueden seleccionar diferentes procedimientos de tramado y diferentes factores de aumento.

Effects contiene los efectos *Color Reduction* (reducción a menos colores), *Greyscale* (cálculo de escala de grises), *Sharpen* y *Soften* (resaltar y suavizar), *Spatial Posterization* (efecto de mosaico) y *Promote to twenty-four Bits* (conversión a TrueColor). *Transform* contiene la posibilidad de reflejar y rotar.

16.14 Calcular geniales gráficos fractales con FractInt 18.2

Para instalar FractInt, copie FRACT182.ARJ a un directorio y desempaquételo. El programa dispone de una extensa función de ayuda. Si a pesar de todo necesitara una documentación, una llamada de FRACTINT MAKEDOC genera un archivo de texto con el nombre FRACTINT.DOC.

FractInt es un generador de fractales, que convence por su gran potencia, rápida presentación y extensas funciones. La alta velocidad FractInt se la debe sobre todo a la aritmética de enteros, que se emplea en vez de los lentos cálculos de coma flotante. Estos últimos también están disponibles para los propietarios de un coprocesador matemático. En el menú principal se explican todas las teclas, que están disponibles en cualquier punto del programa y no sólo aquí. Se puede llegar al menú principal desde cualquier punto del programa mediante [ESC].

Select Video Mode o la tecla [Supr] elige el modo gráfico deseado. Este se puede activar mediante las teclas de cursor o las teclas de función y combinaciones de las mismas (éstas últimas también funcionan en todo el programa). Después de la elección del modo, comienza inmediatamente el cálculo del fractal seleccionado.

Select Fractal Type o la tecla [T] le permite elegir un fractal de una extensa lista, que posteriormente podrá configurar con varios parámetros.

Info about Image o [Tab] muestra informaciones sobre la imagen actual, siempre y cuando se esté calculando una. Una imagen calculada se puede manipular de múltiples formas. Pulsando sobre el botón izquierdo del ratón activará el modo de *Zoom*. Mediante movimientos del ratón se puede desplazar la zona a ampliar, con

el botón pulsado se puede ampliar o reducir la misma y con el botón derecho pulsado se puede rotar. Este zoom no tiene nada que ver con un simple zoom de píxeles de los conversores gráficos. En FractInt se calcula de nuevo la imagen completa ampliada (a plena resolución). Mediante una pulsación sobre [C] se puede activar el modo de *Color Cycling* (y abandonarlo de nuevo con [ESC]). En este modo se inicia una rotación de colores en una dirección determinada, fijada por [+] y por [-], lo que puede crear efectos muy interesantes. Se puede cargar una paleta completamente nueva mediante [L]. Esta función encuentra aplicación en este libro en el Voxel-Spacing, asignándole la paleta *landscap.map* a una nube de plasma.

Save Image to File o [S] guarda (fuera de modo de *Color-Cycling*) una imagen en el disco duro (nombres FRACT001.GIF, etc.). Esta imagen se puede seguir empleando en la potentes herramientas 3D de FractInt:

3-D Transform from File o [3] genera una proyección tridimensional de un archivo GIF cualquiera (por ejemplo una nube de plasma guardada). Después de especificar el archivo fuente y el modo de vídeo a emplear, se presentan amplias funciones de configuración. La siguiente pantalla pregunta por el método de representación de la superficie. Como regla genérica se puede decir que la calidad de la imagen (y con ello el tiempo de cálculo) aumenta cuanto más abajo se encuentre la opción que escogemos. Según el tipo escogido aparecen varias pantallas de opciones que puede confirmar simplemente con [Return].

16.15 Calcular gráficos fantásticos con Povray 2.0

Para la instalación sólo necesita copiar el programa POVIBM.EXE que se autodescomprime, a un directorio de su elección y ejecutarlo. Povray es un Freeware-Raytrace orientado a texto con un conjunto de funciones sorprendente. Se pueden construir escenas complejas como habitaciones completas o paisajes fantásticos con los elementos del lenguaje de Povray. A cada cuerpo se le puede asignar una superficie especial, que es definida por muchos detalles (textura, color, reflexión, etc.). Naturalmente ya se dispone de muchas texturas predefinidas, que representar agua, el cielo o mármol. Povray lee las escenas de un archivo de texto, es decir que el mundo completo con todas sus coordenadas y otras definiciones se ha de escribir a mano con un editor, de forma similar a un lenguaje de programación. Povray "compila" (en este caso realiza un "rendering") de estos archivos. Sería imposible explicar aquí el conjunto de funciones completo (POVRAY.DOC tiene más de i100 páginas!), ya que con el programa Moray existe una cómoda posibilidad de editar escenas. Sólo queremos mencionar un pequeño ejemplo, para demostrar la estructura del lenguaje (muy similar al C):

```
#include "colors.inc"
#include "textures.inc"

camera { location <10,0,0>
         look_at <0,0,0> }

sphere { <0,0,0> , 3
         texture { Red_Marble } }

light_source { <10,10,-10> color White }
```

Los dos comandos *Include* incluyen las definiciones predeterminadas de colores y texturas que se necesitarán a continuación. La instrucción *camera* define la posición del observador (*location*) y el punto que está observando (*look_at*). Existen muchas más instrucciones en el marco de la *camera*, así como de los demás objetos de esta demostración, que se explican extensamente en la documentación. Las coordenadas y los vectores se incluyen en Povray dentro de los signos de mayor/menor, siendo separados por comas.

El primer cuerpo se define por *sphere*. En este caso con centro en el origen (0, 0, 0) y con un radio de tres unidades. Como textura se emplea la predefinida *Red_Marble* (que se puede encontrar en TEXTURE.INC). Finalmente se necesita una fuente de luz, que en este caso se encuentra en las coordenadas (10, 10, -10) y posee un color blanco. Esta escena se calcula (su nombre sería DEMO.POV) mediante

```
POVRAY +IDEMO.POV +X +D +LC:\POVRAY\DOCS
```

Las siguientes opciones se le pueden pasar a Povray (entre otras):

Opción	Significado
+I	Define el archivo fuente.
+X	Activa el control de teclado (cancelación en caso de tecla pulsada).
+D	Activa la representación VGA en la pantalla. Si desea una representación en TrueColor, define entonces *+D0T*.
+V	Visualiza informaciones en la pantalla de texto.
+L	Define la vía de acceso para los archivos Include
+O	Define el archivo de salida (Predefinido: *DATA.TGA*).
+Qn	Determina la calidad (n= 0 - 9).
+W	Indica la anchura de la imagen.
+H	Indica la altura de la imagen.
+C	Continúa un cálculo interrumpido.

16.16 Un estupendo entorno para Povray - Moray 1.3

Para la instalación ha de copiar MORAY13.ZIP a un directorio y desempaquetarlo (PKUNZIP * d). Como el programa no posee ningún tipo de rutina de instalación, las vías de acceso a Povray se han de incluir a mano en el archivo MORAYPOV.CFG. Para ello escriba bajo la opción [*config*] (casi al final) la vía de acceso detrás de *PrintPath10* y *PrintPath20*. Esto puede tener, por ejemplo, el siguiente aspecto:

```
[config]
PrintPath10  'C:POVRAY\'
PrintPath20  'C:POVRAY\'
```

Moray es un entorno gráfico para desarrollar escenas de Povray. Con el ratón se pueden crear objetos, transformarlos y colocarles texturas. La salida a un archivo de texto, como lo necesita Povray, la realiza Moray para usted. Después de ejecutarlo con

```
Moray [Nombre de escena]
```

se pueden ver en pantalla tres vistas de la escena desde un lado, desde delante y desde arriba. Abajo a la derecha se muestra cómo ve la cámara la escena, y cómo la calculará Povray. En las tres ventanas de 2-D se puede realizar un zoom con el botón izquierdo del ratón y la tecla [Alt] pulsados, y un desplazamiento con la tecla [Ctrl]. En el menú del margen derecho se pueden elegir las acciones:

Create: Crea un objeto nuevo: *Cube* (cubo), *Sphere* (esfera), *Cylinder*, *Cone* (cono) o *Torus* son las llamadas *Primitives*. Además también se dispone de los *Sweeps*, que generan un cuerpo tridimensional de una superficie definida. *Rotational* rota una línea definida alrededor del eje z, *Translational* la desplaza a lo largo del eje z, *Conical* genera un cono de ella. Además se puede agrupar conjuntos de objetos (*Group*) y unirlos (*CSG*, se permiten sumas, restas e intersecciones algebraicas de cuerpos).

Pointligh: Crea una fuente de luz, igualmente *Spotlight*, sin embargo con la diferencia de que sólo se ilumina en una dirección determinada.

Edit: Sirve para manipular un objeto. Selección de una textura, mediante *New* un salto al editor de texturas, *Extended Edit* para propiedades especiales del cuerpo en cuestión, etc.

Textures: Abre el editor de texturas. *Create* genera una nueva textura (predefinida). Esta aparecerá desde ahora en la lista de texturas de todas las ventanas de edición, y ha de ser colocada en los objetos correspondientes.

Copy: Copia el objeto seleccionado, y lo transforma simultáneamente.

Files: Sirve para guardar, cargar y exportar (Povray) de la escena. Si esta todavía no tiene nombre, se le ha de asignar uno mediante un clic sobre la línea que hay encima del directorio.

Para poder manipular un objeto, este primero se ha de seleccionar. Para ello se puede hacer un clic sobre el botón *Select*, y por otra parte se puede estirar un recuadro con las teclas izquierda del ratón y [Shift] pulsadas, alrededor del objeto deseado.

Si se encuentra en el menú principal, el objeto puede ser transformado. Para ello elija *SCL* para escalarlo, *USCL* (*uniform scale*) para un escalado uniforme en todos los ejes, *Rotate* para rotaciones o *Trans* para traslaciones. A continuación se puede realizar la función correspondiente con ayuda del ratón o mediante el teclado, en las ventanas 2-D.

La cámara tiene un papel especial: *USCL* modifica en este caso el ángulo de apertura. *Trans* desplaza la posición de la cámara, el punto al que "mira" se puede desplazar después de pulsar sobre [L] (*look_at*). Al revés, se puede volver a la posición con [P] (*Position*).

Epílogo

¡Sí, lo ha conseguido! Ha llegado al final de esta páginas realmente "calientes". Pero este no debe ser el final, sino el principio. En las páginas anteriores ha aprendido con esfuerzo todos los conocimientos que necesita para la entrada a la escena. Unos conocimientos que nosotros tuvimos que obtener a lo largo de muchos años a base de pesado trabajo milimétrico. Seguramente aún no podrá "sacarse de la manga" una demo como UNEATABLE (TC) o WARP (LD). Pero nosotros comenzamos con mucho menos información de la que puede encontrar en esta obra de consulta. Prácticamente es el libro de referencia definitivo para todos los programadores de demos y fundadores de grupos de demo. Una base en la que se puede construir bien y con facilidad. Así que, comience a moverse, espabile y comience su propia producción. Y ya casi es parte integrante de la escena. Solo que, un momento, ¿no falta algo? Si, correcto, ha perpetrado una obra de arte fascinante, ¡pero el mundo aún no lo sabe!

Así que: ¡directamente a la BBS más próxima! A lo mejor no cualquier BBS es el mejor punto de salida para usted, debería ser una del tipo "DEMO ORIENTED". Por ello, a continuación unos cuantos números de algunas BBS del extranjero.

¿Extranjero? ¿Y eso no es demasiado caro? No, ya que en el extranjero frecuentemente obtiene FREE DOWNLOAD como LD-Caller (Long Distance Caller). De modo que: ¡A por ellos!

SOUND SOLUTION BBS
07130 20110
SYSOP: Roger
Gravis Ultrasound en condiciones espectaculares,
kewler Sysop, Demos & GUS-Support

GOLDEN iMAGE BBS
06039 46124
SYSOP: BRiAN
LEGEND DESiGN WHQ, Demos, FAAAST,
ONE OF THE BEST SYSOPS !!!

DARK iLLUSiON
089 36102651
SYSOP: KAOS
ULTRATRACKER WHQ, Demos, Musix

The Firm
040 6482146
SYSOP: Doc BoBo
Demos, Free DL

THE MAGiCAN BBS
0421 4841163
SYSOP: ByTe/BLANK
BLANK WHQ, Demos, KEWL SYSOP

A.C.E. BBS
+33 145 887 548
SYSOP: Gandalf/Infiny
INFINY WHQ, Demos

THE PHOENiX RiSiNG
+1 (806) 655 0712
SYSOP: PHOENiX
Demos

THE SUPREME COURT
+31 1650 51850
SYSOP: STEEL/SPR
SUPREME WHQ, Demos

STAIRWAY TO HAVEN
+45 747 263 03
SYSOP: BIONIC
ViSUAL DREAMS WHQ, Demos

Apéndice

Contenido del CD del libro

El CD contiene gran cantidad de software y ejemplos de programación. Para simplificarle un poco el encontrar los distintos archivos, a continuación detallamos un resumen de la estructura del CD. En el directorio BONUS puede encontrar además diferentes demos gráficas, que seguramente le darán ideas para programas propios.

En el directorio DEMOS se encuentran dos demos: la demo de efectos de *Periscopes* y la megademo de *The CoExistence*. En el directorio RAIDER se encuentra el juego Raider. Raider se programó casi como "sacrificio" especialmente para el entrenador de juegos, por lo que no resulta muy complicado.

Finalmente puede encontrar en el directorio SHARE todas una serie de programas Shareware elegidos especialmente. Todos los programas se emplean preferentemente por los "locos de la escena".

Todos los demás directorios como GRAFICOS, MATES, MEMORY, XMS, FLAT, DMA, NODEBUG, NORESET, CLAVE, PORTS, RTCLOCK, SOUND, VOC, SBMOD, GUSMOD, SPEAKER y TRAINER contienen los programas de ejemplo del libro.

```
Lista de directorios del soporte de datos
E:\
├─BONUS        (Diversas demos gráficas                    )
├─CLAVE        (Rutinas de consulta de clave y manuales    )
├─DEMOS
│  ├─PERISCOP  (Demo de Periscopes                         )
│  └─UNBEATBLE (Demo UNBEATBLE de The Coexistence          )
├─ENTRENAD     (Programación de entrenadores de juegos     )
├─GRAFICOS     (Ejemplos de programación de gráficos del libro)
├─MATEMAT      (Rutinas matemáticas alternativas           )
├─MEMORIA
│  ├─DMA       (Programación del Direct Memory Access      )
│  ├─FLAT      (Modelo de memoria Flat del procesador      )
│  └─XMS       (Gestión de memoria XMS y EMS               )
├─NODEBUG      (Impedir debugging                          )
├─NORESET      (Impedir arranque en caliente               )
├─PUERTOS      (Programación directa de los puertos        )
├─RAIDER       (Simple juego de disparos para el entrenador )
├─RTCLOCK      (Programación RTC                           )
├─SHARE
│  ├─ALCHEMY   (Image Alchemy v1.7                         )
│  ├─ARJ       (Programa compresor ARJ 2.41                )
│  ├─DBG       (Debugger DBG                               )
│  ├─FRACTINT  (Fractint v18.2 para DOS                    )
│  ├─GWSHARE   (Game Wizard                                )
│  ├─GWSHIN    (Graphic WorkShop para Windows              )
│  ├─HEXCALIB  (Editor Hexcalibur para DOS                 )
│  ├─MORAY     (Entorno gráfico para POVRAY                )
│  ├─PICVIEW   (Conversor y visualizador de gráficos       )
│  ├─PKLITE    (PKLITE, compresor para archivos EXE        )
│  ├─PKZIP     (PKZIP, compresor de archivos               )
│  ├─POVRAY    (Raytracer POVRAY v2.0                      )
│  ├─PSP       (Paint Shop Pro 2.0 para Windows            )
│  ├─ST3       (Scream Tracker 3.0, Editor MOD             )
│  ├─TOSCOPY   (Programa copiador TOS-COPY                 )
│  ├─UMP       (Programa descompresor y desencriptador     )
│  ├─VOC386    (VOCEDIT 386, Editor VOC                    )
│  └─VPIC      (Conversor y visualizador de gráficos       )
├─SONIDO
│  ├─GUSMOD    (MOD-Player para la Gravis Ultrasound       )
│  ├─SBMOD     (MOD-Player para la SoundBlaster            )
│  └─VOC       (VOC-Player para la SoundBlaster            )
└─SPEAKER      (Reproducción de sonido por el altavoz      )
```

Tabla ASCII

Dec.	Hex.	Carácter	Dec.	Hex.	Carácter	Dec.	Hex.	Carácter	Dec.	Hex.	Carácter
0	00		32	20		64	40	@	96	60	
1	01	☺	33	21	!	65	41	A	97	61	a
2	02	☻	34	22	"	66	42	B	98	62	b
3	03	♥	35	23	#	67	43	C	99	63	c
4	04	♦	36	24	$	68	44	D	100	64	d
5	05	♣	37	25	%	69	45	E	101	65	e
6	06	♠	38	26	&	70	46	F	102	66	f
7	07	•	39	27	'	71	47	G	103	67	g
8	08	◘	40	28	(	72	48	H	104	68	h
9	09	○	41	29	)	73	49	I	105	69	i
10	0A	◙	42	2A	*	74	4A	J	106	6A	j
11	0B	♂	43	2B	+	75	4B	K	107	6B	k
12	0C	♀	44	2C	,	76	4C	L	108	6C	l
13	0D	♪	45	2D	-	77	4D	M	109	6D	m
14	0E	♫	46	2E	.	78	4E	N	110	6E	n
15	0F	☼	47	2F	/	79	4F	O	111	6F	o
16	10	►	48	30	0	80	50	P	112	70	p
17	11	◄	49	31	1	81	51	Q	113	71	q
18	12	↕	50	32	2	82	52	R	114	72	r
19	13	‼	51	33	3	83	53	S	115	73	s
20	14	¶	52	34	4	84	54	T	116	74	t
21	15	§	53	35	5	85	55	U	117	75	u
22	16	▬	54	36	6	86	56	V	118	76	v
23	17	↨	55	37	7	87	57	W	119	77	w
24	18	↑	56	38	8	88	58	X	120	78	x
25	19	↓	57	39	9	89	59	Y	121	79	y
26	1A	→	58	3A	:	90	5A	Z	122	7A	z
27	1B	←	59	3B	;	91	5B	[	123	7B	{
28	1C	∟	60	3C	<	92	5C	\	124	7C	|
29	1D	↔	61	3D	=	93	5D	]	125	7D	}
30	1E	▲	62	3E	>	94	5E	^	126	7E	~
31	1F	▼	63	3F	?	95	5F	_	127	7F	⌂

Dec.	Hex.	Carácter	Dec.	Hex.	Carácter	Dec.	Hex.	Carácter	Dec.	Hex.	Carácter
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	η
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Dec.	Hex.	Character	Dec.	Hex.	Character	Dec.	Hex.	Character	Dec.	Hex.	Character
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Índice alfabético

A86 ... 632
Acceso ... 426
Acceso 32 Bits ... 116
Acceso byte ... 116
Acceso de escritura ... 91, 156, 273
Acceso de lectura ... 118, 156, 273
Acceso Doubleword ... 115
Acceso Word ... 116
Adaptación ... 25
Adaptación de formato ... 76
Ajuste de fino ... 153
Alarm Interrupt ... 415
Alarm Interrupt Enable ... 414
Alfabeto ... 65, 133
Algebra ... 229
Algoritmo ... 65
Algoritmo de compresión ... 65, 79
Algoritmo de desempaquetado ... 66
Algoritmo de relleno ... 260
Algoritmo lineal ... 26, 238
Algoritmo Quicksort ... 276
All-Mask-Register ... 405
Amiga ... 135, 577
Amplitud de paso ... 175, 177
Analizar ... 324
Anchura ... 128
Anchura de línea ... 128
Ángulo ... 273, 277, 283
Animación ... 135
Apagar ... 632
Aproximación ... 22, 28, 282
Archivo ... 616
Archivo COM ... 634
Archivo fuente ... 65
Archivo objeto ... 634
Área ... 237, 273
Áreas de valor fijo ... 280
Arista de polígono ... 270
Aristas de polígono, tridimensionales ... 287
ARJ 2.41 ... 615
Arpegio ... 572, 588
Arquitectura de bus ... 115
Arreglo ... 569, 575, 576, 585
ASSUME ... 633
ATC ... 156
Attribute Controller ... 147
Automodificación ... 271
Autotest ... 417
Ayuda ... 627
Azul ... 63
Backup ... 619
Balance ... 425
Barras ... 158, 161
Barras Copper ... 158
Base de cálculo ... 145
Base de tiempo ... 414
Base Memory ... 418
Base-Port ... 570
Basura de bits ... 77, 116
Batería ... 415
Bibliotecas de idiomas ... 19
BIOS ... 60, 87, 403
Bit de direcciones ... 384
Bitmap de textura ... 283
Bits divisores ... 412
Bloque de ampliación ... 77
Borde ... 144
Borde de pantalla ... 129, 200
Borrado ... 616
Borrar ... 131, 616, 631
BPM ... 582, 597
Breakpoint ... 323, 346
Bresenham ... 26, 238
Brillo ... 57, 170, 277, 279, 281, 283
Brillo máximo ... 170
Bucle de línea ... 272
Bucle de representación ... 271
Bucle principal ... 187
Buffer ... 581, 631
Buffer anular ... 407
Buffer de destino ... 199
Bus ISA ... 135
Búsqueda ... 326
Byte contador ... 88
Byte de atributos ... 61
Byte de control ... 280
Byte de longitud ... 65
Byte de velocidad ... 418
Bytes de estado ... 397
C ... 19
Cadena ... 25, 127
Cadena de caracteres ... 65
Cadenas parciales ... 129
Caída del sistema ... 69
Cálculo 3D ... 233
Cálculo de ángulos ... 278
Cálculo de coma fija ... 21
Cálculo de coma flotante ... 21
Cálculo en cadena ... 25

Cálculos de sonido ... 160
Campos parciales ... 276
Canal ... 395, 396, 405, 588
Canales ... 570
Cantidad ... 271
Cantos ... 237, 257
Carácter ... 61
Caracteres especiales ... 134
Cargador ... 146
Cargar ... 569, 630
Cascada ... 404
Cathode Ray Tube Controller ... 91
Ceros ... 158
Cifras decimales ... 21
Clave ... 618
Clear-Code ... 66
Cli ... 383, 403
Clipping ... 166
CMOS-RAM ... 412
Código fuente ... 323
Color de mezcla ... 257, 258
Color de relleno ... 283
Color del borde ... 161
Color destino ... 178, 181
Color fuente ... 179
Color inicial ... 178
Color primario ... 57, 280, 283
Colores ... 258
Comando ... 396
Combinación ... 178
Combinatoria ... 178
Comercial ... 619
Comienzo de la memoria ... 150, 151
Command-Port ... 426, 428
Comodines ... 616
Compatibilidad ... 116
Componente de color ... 63, 198
Componentes ... 173, 230
Compresión RLE ... 79
Comprimido ... 146
Comprimir ... 65, 77
Comprobación de coordenadas ... 59
Comprobación de rangos ... 386
Concepto de página ... 119
Condición de interrupción ... 169
Configuración ... 415
Configuración del sistema ... 415
Conmutación ... 383
Conmutar ... 276
Conmutativo ... 237
Constante tipificada ... 128
Construcción de pantalla ... 60, 115, 280
Contador de caracteres ... 129
Contador de columnas ... 129
Contador de direcciones ... 407
Contador de líneas de trama ... 160
Contenido de imagen ... 153
Contenido de la memoria ... 621
Contenido de pantalla ... 58
Contenidos de registro ... 136
Control-Register ... 396
Controlador VESA ... 119
Controladora de vídeo ... 159
Controlar ... 158
Conversión de direcciones ... 116
Conversor AD/DA ... 425
Convertir ... 640, 642
ConvertString ... 577
Coordenada 3D ... 289
Coordenada destino ... 273
Coordenadas ... 61, 120, 280
Coordenadas de pantalla ... 283
Coordenadas fuente ... 237
Coordenadas mundiales ... 237
Copiar ... 158, 384
Copias ... 120
Copper ... 160
Coprocesador ... 21, 29
COPY ... 639
Copyright ... 64
Coseno ... 28, 237, 277
CPU ... 61, 418
Criterio de ordenación ... 274, 275
CRTC ... 91
Cuadrado ... 144, 282
Cuerpo de vidrio ... 257, 273
Cut Voice ... 597
Chain-4 ... 116
Chaining ... 60
Chip DSP ... 425
Chip mezclador ... 425
Chip temporizador ... 395

D86 ... 635
DAC ... 87, 164, 171, 173
Dallas 1287 ... 412
Date Mode ... 415
Datos de color ... 63
Datos de imagen ... 185
Datos de trama ... 77
Daylight Savings Enabled ... 415
Debugger ... 323, 635
Debugging ... 150
DEC ... 633
Decodificación ... 77
Decrement ... 345

Dedo ... 136
Definición de área ... 243
Definición del mundo ... 284
Degradado de color ... 191, 279
Degradado de grises ... 61
Derechos de acceso ... 382
Descarga ... 148
Descompresión ... 66
Descomprimir ... 65
Descripción de imagen ... 65
Descripción de registro ... 136
Descriptor ... 382
Desempaquetar ... 616, 617, 619, 620
Desenmascarar ... 122, 405
Desplazado ... 20
Desplazamiento a la derecha ... 139
Desplazamientos de bits ... 138
Desplazamientos de memoria ... 135
Desplazar ... 118
Desvanecimiento ... 185
Determinación de ángulos ... 278
Determinación de valores ... 29
Determinante principal ... 286, 287, 291
Determinantes laterales ... 291
Diafragma ... 173, 178
Diagonal ... 147
Diferencia ... 281
Digitalizar ... 630
Dimensión ... 229
Dirección ... 61, 148, 230, 382, 394, 408
Dirección base ... 425
Dirección de inicio ... 88, 121, 123
Dirección de inicio de pantalla ... 191
Dirección de Port ... 89, 137, 156
Dirección destino ... 294
Dirección X ... 148
Dirección Y ... 148
Direccionamiento ... 60
Directivas ... 20
Directorios ... 617
Display Enable ... 159, 165
Disco duro 1 ... 418
Disco duro 2 ... 418
Discos duros ... 417
Disk timeout ... 395
Dispositivos de salida ... 233
Distancia de salto ... 128, 177
Distancia ocular ... 255
Distorsiones ... 135
Dividido ... 283
División ... 20, 23
División de resolución ... 120
DMA ... 404
DMA-Controller ... 404
DMA-Count-Register ... 409
DOS ... 382
Double ... 148
Double-Buffer ... 407
Double-Scan ... 59, 120, 139
Double-Scan Enable ... 156
Doubleword ... 47, 116
DRAM-Refresh ... 395
DSP ... 425
Duplicación ... 149
Duplicación de línea ... 139
DWord ... 116, 344

Eco ... 632
Ecuaciones ... 287
Ecualizador ... 574
Editor ... 333
Efecto ... 57, 87, 115, 572, 585, 588, 628, 632
Efecto Pyro ... 144
Eje de coordenadas ... 236
Ejecutable ... 619
Electrones ... 58
EMM386 ... 387
Emplear ... 631
Emulación ... 59
En contra de las agujas del reloj ... 273
Enable Chain4 ... 116
End ... 571
Enmascarable ... 391
Enmascaramiento ... 131, 139
Ensamblador ... 19, 345, 393, 632
Entero ... 21
Entorno ... 242, 283
Entrada de alfabeto ... 77
Entradas ... 602
Entradas de paleta ... 57, 178, 181
Entradas por teclado ... 19
Entrenador ... 344, 347
Entrenador de juegos ... 344
EOI ... 392
Equipamiento ... 417
Escalado ... 232, 235
Escribir ... 88
Escritura de coma fija ... 21
Esfuerzo de cálculo ... 234
Espacio de direcciones ... 60
Espacio de disco ... 133, 152
Estado ... 570
Estocástica ... 178
Estructura ... 60, 158, 195, 258, 345, 633
Estructura de superficie ... 243
Execution Pipe ... 383

Existe ... 616
Exponentes ... 20
Extended Memory ... 418
Extension-Block ... 63, 64

Factor ... 22
Factor decimal ... 22
Fakemode ... 398
Falta de memoria ... 148, 618
Feature-Byte ... 418
Fecha ... 413, 620
Figuras ... 200, 229
Filtro de ruido ... 632
FillPuf ... 76
Fin ... 571, 573, 619
Fin de línea ... 88
Final de paleta ... 181
Fine Sliding ... 595, 596
Físico ... 61
FLAG ... 391
Flat ... 381
Flecha ... 229
Flip-Flop ... 409
Fondo ... 115, 131, 159
Fondos de un sprite ... 121
Formación de bloques ... 179
Formato BCD ... 413, 415
Formato de bloques ... 65
Formato de contador ... 396
Formato gráfico ... 130
Formato PC ... 577
FractInt 18.2 ... 646
Frecuencia ... 571, 582, 593, 631
Freemem ... 388
Fuego ... 195
Fuente ... 349
Fuente de luz ... 277
Funciones de raíz ... 28
Fundido ... 632

GDC ... 273, 276
GDT ... 382, 383
Genlock ... 159
Geometría ... 229
Gestión de memoria ... 146
GetIntvec ... 393
GetPhysByte ... 76
GIF ... 62, 67, 124, 146
GIF-Loader ... 146, 171
GIFClose ... 75
GIFOpen ... 75
GIFRead ... 75
GIFSeek ... 75
Global Color Map ... 63
Global Descriptor Table ... 382
Grabber ... 87
Graphic Workshop for Windows ... 645
Guardar ... 630

Hacia atrás ... 632
Happen ... 146
Hardcopy ... 87
Hardware ... 391
Herramienta ... 319, 324
Hex-Editor ... 333
HiColor ... 57, 158
High-Byte ... 280
Homogéneo ... 277
Hora ... 413
Hora de alarma ... 415
Hora del día ... 395
Hora del sistema ... 395
Horizontal ... 144, 147
Horizontal-Sync Start ... 165
Horizontal-Sync-Start-Register ... 166
Horizontal-Timing ... 166
Hotkey ... 86

IBM-VGA ... 115
IDB ... 77
Identificación de área ... 275
Identificador ... 77
Identificador de terminación ... 63
Iluminación ... 277
Iluminar ... 319
Image Alchemy 1.7 ... 642
Image Descriptorblock ... 63
Imagen de fondo ... 121, 139
Imagen destino ... 181, 183
Imagen original ... 180
Imágenes ... 115
Imágenes parciales ... 144
Imprecisiones ... 25
INC ... 633
Incompatibilidad ... 121
Index-Mode ... 156
Índice ... 156
Información de colores ... 257, 280
Información de longitud ... 230
Información de profundidad ... 234, 274, 275
Informaciones ... 570, 602, 630
Informaciones de instrumento ... 577
Informaciones de segmento ... 382
Informaciones de tiempo de reproducción 574
Informático ... 271
Inicia_Tono ... 572

Inicialización 588, 602
Inicializar 287
Inicio 274, 573
Inicio de bucle 28
Inicio de pantalla 130, 139, 142, 144, 149
Inline-Assembler 386
Input-Status-Register 159
Instalación 347
InstNr 571
Instrumento 572, 576, 577, 585, 628
Integer 21
Integridad 616
Interlace 64
Interrupción de actualización 414
Interrupción de línea de trama 159
Interrupción enmascarable 403
Interrupción original 394
Interrupciones 391
Interrupt 19, 159, 165, 383, 391, 394, 582
Interrupt-Request-Flag 415
Interruptor de red 136
Interruptor DIP 159
Invertido 274
Invisible 123
Iret 391

Juego de caracteres 123, 133
Juego de chips 136
Juego de rol 195

Last-Byte 88
Latches 119, 276
Leer 88
Lenguajes de alto nivel 19
Ley distributiva 20
Lgdt 383
Licencia 619
Limitaciones 621
Line-Compare 137
Line-Compare-Register 87
Línea 120, 260, 575
Línea de comandos 602
Línea de estado 137
Línea de relleno 289
Linear Counter 145
Linear Starting Address 87, 139, 142, 153
Linear-Starting-Address-Register 139
Líneas de direcciones 61
Líneas de imagen 64
Líneas de limitación 260
Líneas de trama 158
Líneas gráficas 141
LoadGIF 148, 150
LoadGIF_Pos 150
Local Color Maps 63
Logical Screen Descriptor Block 63
Lógica de prioridades 402
Longitud 230, 384, 385, 573, 577
Longitud de línea 288
Longitud de tema 576, 585
Loop_Start 573
Looping 571, 572, 573, 585
Lower-Page-Register 408
Lugar 230
Lugar decimal 21
LZW 62

LLamas 197

Macros 20
Madera 283
Manipulaciones 136
Manipulaciones de registro 58, 136
Mapas de bits 283
Mar de llamas 199
Marco de pantalla 161
Más alto 632
Mas bajo 632
Más lento 632
Más rápido 632
Máscara 161, 164, 273
Máscara de prioridades 161
Mask-Bit 405
MASM 632
Matemáticas 178
Materiales 283
Matrices de rotación 235
Máximo 271
Maximum Row Address 138, 139, 156
Mayúsculas 134
Memoria 387, 404, 601
Memoria 627
Memoria de ampliación 418
Memoria de juego de caracteres 61
Memoria de pantalla 57, 115, 116, 189, 273, 276
Memoria de vídeo 115
Memoria del sistema 60
Memoria gráfica 146
Memoria intermedia 119
Memoria principal 130, 146, 384, 385, 387
Memory Mode 116
Memory-Tables 621
Mempos 571, 573, 581
Mermas de velocidad 21
Metal 283

Metamorfosis ... 178
Método de compresión ... 66
Método de direccionamiento ... 121
Método RLE ... 80
Mezcla ... 159
Mínimo ... 270
Mitades ... 152
MOD ... 624
MOD_Patternsize ... 576
Mode 13h ... 87
Mode X ... 87, 115
Modelos de alambre ... 237
Modificación de registro ... 166
Modinfo ... 574
Modo ... 60
Modo 13h ... 115
Modo bloque ... 407
Modo de cascada ... 407
Modo de petición ... 407
Modo Double-Scan ... 156
Modo gráfico ... 57, 62, 87, 157
Modo singular ... 407
Modo texto ... 117, 132, 152, 157
Modos de funcionamiento ... 395
Moray 1.3 ... 649
Move ... 185
Movimiento ... 135, 141
Movimiento irregular ... 140
Multiplicación ... 20, 22, 231
Multiplicación escalar ... 232
Multiplicando ... 20
Mundos ... 277
Muting ... 570

NMI ... 403
Nombre ... 573
Nombre de archivo ... 602
Nombre de directorio ... 617, 618
Nombre de instrumento ... 577
Norton ... 320
Note & Volume sliding ... 592
Número ... 91
Número de índice ... 412
Número de plane ... 121
Número de bits ... 20
Número de bloque ... 178
Número de colores ... 62, 180
Número de coma fija ... 21, 22, 25
Número de línea ... 139
Número de píxeles ... 64
Número de punto ... 118
Número de registro ... 91
Número de versión ... 79, 428
Números ... 134
Números de coma flotante ... 21, 25

Objetos ... 257
Observador ... 200
Odd/Even-Mode ... 116
Offset ... 285, 384, 385, 408, 577
Offset de memoria ... 121, 139
Offset de tabla ... 169
Operaciones de cálculo ... 21
Operaciones de lenguajes de alto nivel ... 21
Operando ... 572
OPL-Chip ... 425
Optimización ... 19
Optimizado ... 229
Orden ... 146, 237, 273, 617
Orden de ordenación ... 63, 64
Ordenación ... 273, 274
Ordenar ... 274
Organización de la memoria ... 60
Original ... 395
OS/2 ... 19
Overflow-Register ... 138, 139
Overlay ... 164, 619

Página ... 408
Página de reserva ... 121
Páginas de pantalla ... 115, 119, 121, 144, 179
Painter's Algorithm ... 274
PaintShop Pro ... 643
Paisaje ... 200
Paleta ... 57, 63, 276
Paleta de colores ... 60, 62
Paleta de destino ... 173, 181
Paleta estándar ... 61
Paleta local ... 64
Panning ... 153
Panning-Register ... 153
Pantalla ... 576
Pantalla de ayuda ... 615
Pantalla parcial ... 152
Parámetro ... 616
Parity ... 403
Parpadeo ... 115, 136, 140, 179
Parte de decimal ... 21, 22, 27
Parte ensamblador ... 69
Parte entera ... 21, 22, 27
Pascal ... 19, 128
Pasa altos ... 632
Pasa bajos ... 632
Pascal-String ... 577
Paso ... 331
Patrón ... 574, 575, 585, 595, 628

Pattern Break ... 595
Pausa ... 583
PC-Speaker ... 395
PCX ... 65, 79
PCXShift ... 87
Pendiente ... 21, 26, 271
Periferia ... 404
Periodic Interrupt Enable ... 414
Período de oscuridad ... 166
Perspectiva de punto de fuga ... 229, 235, 237
PIC ... 402
Pico ... 574
PIT ... 395
Píxel ... 57
Pixel-Color-Value-Register ... 165
PKLITE ... 320, 619
PKZIP ... 615
Plane ... 61, 117, 273, 294
Plane de destino ... 273
Planos ... 61
Plataforma de destino ... 136
Polígono ... 260
Port ... 156, 396
Port de datos ... 426, 428
Portamento ... 589, 590
Posición ... 61, 79, 148, 191, 572, 575, 594
Posición de paleta ... 185
Posición inicial ... 581
Posiciones de caracteres ... 127
Postfix ... 77
Potencias de dos ... 20
Povray 2.0 ... 647
Precisión ... 21, 288
Precisión de cálculo ... 23, 31
Prefetch-Queue ... 383
Prefijo ... 77
Preguntas de seguridad ... 59
Presentación ... 132
Primer plano ... 159
Prioridad ... 402
Problema de memoria ... 381
Procedimiento de copiado ... 130
Procedimiento de representación ... 243
Procedimientos ... 20
Procesador ... 382
Proceso de copia ... 119
Producto escalar ... 278, 283
Producto vectorial ... 233, 277, 282
Profundidad de color ... 57, 67
Programa copiador ... 638
Programa de dibujo ... 133
Programa principal ... 243
Programación de demos ... 398
Programación de gráficos ... 57
Programación de interrupciones ... 391
Programas compresores ... 62
Protected Mode ... 382
Protection-Bit ... 91, 138, 166
Proyección paralela ... 233, 234, 237
Proyectado ... 283
Proyectos ... 633
Public ... 21
Puntero ... 146
Puntero ... 60, 145, 272, 382
Puntero de destino ... 187
Puntero de superficie ... 275
Punto ... 260
Punto final ... 274, 289
Punto inicial ... 260, 289
PV 2.41 ... 640

QEMM ... 387
Quicksort ... 275

Radicando ... 282
Ralentización ... 140
RAM ... 142, 418
RAM de Vídeo ... 385
Rampas de brillo ... 229
Rango de definición ... 31
Rango de destino ... 385
Rango de valores ... 120
Rangos ... 137
Rayo catódico ... 58, 115, 136, 145
Rayo de luz ... 277
Rayo de trama ... 164
Rayos ... 234
Raytracing ... 233, 277
READ ... 639
Read-Mode ... 116
Real-Mode ... 382
Realidad ... 257
Rebasamiento ... 21, 22, 77, 412
Rebase de segmento ... 146
Recortar ... 630
Recorte de pantalla ... 58
Recta ... 26
Rectas ópticas ... 273
Recursivo ... 276
Redes Mailbox ... 62
Reducir ... 140
Reflexión ... 227
Registro ... 136, 394, 396, 405
Registro A ... 414
Registro B ... 414
Registro C ... 415

Registro CRTC ... 87
Registro D ... 415
Registro de banderas ... 415
Registro de datos ... 91, 156
Registro de entrada de contador ... 397
Registro de estado ... 396
Registro de estado A ... 412
Registro de estado B ... 412, 413
Registro de índice ... 91
Registro de paleta ... 181
Registro de rebasamiento ... 138
Registro de segmento ... 146
Registro individual ... 89
Reglas básicas matemáticas ... 21, 22
Reglas de compatibilidad ... 19
Regulación ... 425
Relativo ... 230
Releer ... 397
Relocation ... 619
Rendering ... 234
Renovar ... 617
Reparar ... 616
Repeat ... 631
Repeticiones de bytes ... 80
Repeticiones de caracteres ... 88
Repetido en bucle ... 570
Representación en colores falsos ... 140
Representaciones de sprites ... 19
Reproducir ... 569, 601, 627, 631
Reprogramar ... 136
Requerimientos de memoria ... 62, 629
Reset ... 383, 426
Reset-Block ... 180, 181
Reset-Port ... 426
Residente ... 19
Resolución ... 62, 63
Resolución vertical ... 120
Resta ... 22, 231
Restaurar ... 131
RestByte ... 76
Resultado intermedio ... 283
Retorno del rayo ... 115
Retrace ... 58
Retrazado horizontal ... 159, 165
Retrazado vertical ... 159, 398
Retriggering ... 572, 596
Reverberación ... 632
RLE ... 65
RMEM ... 384, 386, 390
Rojo ... 63
Rotación ... 227, 235, 236, 241, 283
Rotación de paleta ... 190
RTC ... 412, 413, 415, 419
RTC-RAM ... 412
Rutina Detect ... 427
Rutinas de ordenamiento ... 388

S3M ... 624
Sacar raíces ... 22
Salida de caracteres ... 132
Sampling ... 629
Sample ... 569, 627
Sample-Offset ... 593
Samplear ... 287
SB16 ... 428
Scream Tracker ... 624
Screenshot ... 86
Scrolling ... 144, 147, 151
Secuenciadores ... 116
Segment-Overrides ... 633
Segmento ... 384, 385
Segmento cero ... 382
Segmento de código ... 346
Segmento inicial ... 187
Segmentos de datos ... 346
Seguir rotando ... 294
Selección ... 61
Selectores ... 382
Selectores de segmento ... 115
Seno ... 28, 237
Set Speed ... 597
SET-Bit ... 412
SetIntvec ... 393
SetStart ... 151
Shift ... 118
Shutdown-Status-Byte ... 416
Signatura de formato ... 79
Signo ... 271
Silbido ... 136
Silencio ... 632
Símbolo del sistema ... 615
Simplificación ... 277
Sincronización ... 140, 165
Sincronización horizontal ... 169
Sincronizado ... 171
Single-Mask-Register ... 405
Síntests FM ... 425
Sistema de ecuaciones ... 286
Sistema operativo ... 391
Sistemas de bus ... 116, 135
Sliding ... 572
Sobreimprimir ... 188
Sobrescrito ... 619
Software ... 391
Solapar ... 161
Sombreado de fuente de luz ... 229, 280

Sonido espacial ... 578
Speed ... 597
Split ... 138
Split-Line ... 139
Split-Screen ... 137, 139, 140, 151
Splitting ... 151
Sprite ... 58, 132
Square Wave Enable ... 414
Starscroller ... 59, 60
Sti ... 403
Subdirectorios ... 618
Suma ... 22, 231
Suma de vectores ... 235
Sumado ... 231, 282
Super-VGA ... 119, 138
Superficies de vidrio ... 276
Supresión ... 273, 275
Supresión de caras ocultas ... 273
Sustituir ... 273, 631

Tabla ... 29, 127
Tabla de coordenadas ... 271
Tabla de senos ... 30, 166, 169, 241
Tamaño ... 79, 382
Tamaño de bloques ... 65
Tamaño del sprite ... 58
Tareas de control ... 158
Tarjeta VGA ... 385
Tasa de compresión ... 66
TASM ... 632
Temperatura ... 197
Temporizador ... 582
Temporizador del sistema ... 395
Terminar ... 601
Textos fuente ... 633
Texturas ... 283, 285
TEXTURE.INC ... 286
Tiempo ... 620
Tiempo de cálculo ... 152, 234
Tiempo de reproducción ... 619
Tiempo real ... 277
Timing ... 116, 120, 166
Timing Sequenzers ... 130
Tipo ... 418
Título ... 574
Tolerancia ... 632
Tone Portamento ... 590
Tono ... 395, 572, 585
Tono_destino ... 572
TOS-Copy ... 638
Trabajo ... 396
Tramo ... 280
Transferencia ... 404, 406
Transferencia ... 407
Transferencia de escritura ... 407
Transferencia de lectura ... 407
Transferir ... 404
Transformaciones ... 229
Traslación ... 235
Tremolo ... 575, 593
Tridimensional ... 200, 283
TrueColor ... 57, 158
TSR ... 164, 344

UIP-Bit ... 414
Ultima Underworld ... 283
Ultra_Mem2Gus ... 581
Unidad de destino ... 639
Unit ... 127, 569
Universalidad ... 77
Unpack ... 320
Update ... 617
Update Ended Interrupt Enable ... 414
Update Ended Interrupt-Flag ... 415
Utilidades ... 320

Valor de color ... 165, 257
Valor de control ... 280
Valor de máscara ... 161
Valor de registro ... 120
Valor estándar ... 414
Valor inicial ... 152
Valor medio ... 199, 274
Valor original ... 78, 153
Valles ... 200
Variable BYTE ... 344
Variables ... 19, 394
Variables de línea ... 272
Variables de registro ... 152
Varios Archivos ... 618
Vector ... 229
Vector de luz ... 277, 283
Vector de traslación ... 235, 236
Vector normal ... 277, 281, 282, 283
Vectores de lugar ... 230, 235
Vectores de superficie ... 280, 281, 286
Velocidad ... 59, 77, 382, 572, 575, 619
Velocidad de acceso ... 115
Velocidad de cálculo ... 21, 22
Velocidad de datos ... 135
Velocidad de descompresión ... 62
Velocidad de ejecución ... 137
Velocidad de procesamiento ... 80
Velocidad de representación ... 238
Ventaja de velocidad ... 22, 116
Ventana ... 132, 142

Verde 63
Verify 407
Versiones de Sound Blaster 428
Vertical 144, 147
Vertical-Retrace 58, 157, 171, 172
Vértices 237, 242
VESA 119
VESA Local Bus 115
Vibración 575
Vibrato 572, 575, 591, 593
Vibrato & Volume sliding 592
Vídeo 159
Vigilancia 135
Vigilar 324
Virtual 149
Virus 320
Visualización 576
Visualizador de archivos gráficos 639
VOC 386 ..... 629, 630
Voces_MOD 574, 576
Volumen 425, 571, 573, 575, 586, 594
Voxel-Spacing 200
VPIC 6.1 639

WaitRetrace 540
Waitstates 122
Watches 331
Windows 19
Word 116, 344
WRITE 639
Write Plane Mask 116
Write-Mode 116, 159

XMS 381, 384, 387

Zona fuente 384
Zoom 631

CP10/E1/R1/98

Esta edición se terminó de imprimir en mayo de 1998. Publicada por ALFAOMEGA GRUPO EDITOR, S.A. de C.V. Apartado Postal 73-267,03311 México, D.F. La impresión y encuadernación se realizaron en los talleres de IMPREANDES PRESENCIA S.A. Santafé de Bogotá, D.C. - Colombia, el tiro fue de 1000 ejemplares.

PC al límite

PROGRAMACION AVANZADA

Funciones, técnicas y secretos de programación no documentados

Conozca los trucos y secretos de los "gurús" de la programación. Un libro obligado para quienes desean entrar al mundo secreto de la programación de aplicaciones multimedios que incluyan gráficos y sonido. Aprenda los trucos de programación no documentados, de los profesionales, en Ensamblador y Turbo Pascal.

Usted también podrá ser un "gurú" de la programación

Se explican técnicas que hasta ahora sólo iban "de boca en boca". Emplee estos conocimientos y conviértase también en un programador experimentado. Comprenderá la estructura de los gráficos Voxel-3D de los actuales super juegos y cómo funcionan los más efectivos mecanismos de protección de software. Los autores le mostrarán cómo puede conseguir impresionantes efectos gráficos, como fuego o llamas, añadiéndoles sonido para crear fantásticos demos.

Protección de programas:

- Con claves de acceso seguras
- "Bloqueando" depuradores

Trucos de programación no documentados:

- Programación de entrenadores de juegos (CHEATS)
- Programas más rápidos mediante optimización
- Trucos para programar los chips de DMA, reloj y temporizador

Memoria sin límite:

- ¡Con el modelo de memoria FLAT hasta los 4 GBYTES!

Sonido sin fin...

- MOD-Player para las tarjetas SoundBlaster
- MOD-Player para la Gravis Ultrasound

Requerimientos del sistema para el CD:

- PC 386 o superior
- 4 Mbytes de memoria
- Unidad de CD-ROM
- Tarjeta VGA

Efectos gráficos que dejan atónito

- Emplear adecuadamente el Texture-Mapping
- Las fantásticas posibilidades de la programación de objetos vectoriales "Glass-Vectors"
- Efectos de llamas y fuego
- Programas propios con Voxel-Space-Graphics

En el Disco Compacto:

- Todos los ejemplos y listados del libro
- Útiles herramientas, "demos" y los programas shareware preferidos de los mejores programadores.

ISBN 970-15-0085-7

9 789701 500859

Alfaomega Grupo Editor

Diseño de la cubierta: Mariana López Benítez